LEC 22

CSE 373

Topo Sort & Reductions

BEFORE WE START

How many total pivots would Quick Sort need for the divide step on this array if we choose the pivot as:

a) First element
b) Median of the first, middle, and last elements

11	2	9	3	8	5	4

pollev.com/uwcse373

Instructor: Aaron Johnston

TAs: Timothy Akintilo, Brian Chan, Joyce Elauria, Eric Fan, Farrell Fileas, Melissa Hovik, Leona Kazi, Keanu Vestil, Siddharth Vaidyanathan, Howard Xiao

The Final Stretch

You are here

You're almost there! Here's what's coming up in the last week of the quarter:

					FRI **Topo Sort, Reductions** (Last day of content for Exam II) **TA-Led Industry Panel 4:30!**	**SAT**
SUN	**MON** **Tries** (Guest Lecture: Eric Fan!!) **EX4 Due**	**TUE**	**WED** **Course Wrap-Up** **P4 Due**	**THU** EX4 Late Cutoff	**FRI** Exam II Released **Exam II OH**	**SAT** **Exam II Due** P4 Late Cutoff Extra Credit Due

Eternal Mastery of Data Structures →

Learning Objectives

After this lecture, you should be able to...

1. Implement Quick Sort, derive its runtimes, and implement the in-place variant
2. Define a topological sort and determine whether a given problem could be solved with a topological sort
3. Write code to produce a topological sort and identify valid and invalid topological sorts for a given graph
4. Explain the makeup of a reduction, identify whether algorithms are considered reductions, and solve a problem using a reduction to a known problem

Lecture Outline

- Comparison Sorts
  - *Review* Sorting Overview
  - In-Place Quick Sort
- Topological Sort
- Reductions
  - Definitions
  - Examples

STRATEGY 1: ITERATIVE IMPROVEMENT

Insertion Sort

WORST $\theta(n^2)$

BEST $\theta(n)$

Simple, stable, low-overhead, great if already sorted.

IN-PLACE STABLE SPACE $\theta(1)$

Selection Sort

WORST $\theta(n^2)$

BEST $\theta(n^2)$

Minimizes array writes, otherwise never preferred.

IN-PLACE SPACE $\theta(1)$

STRATEGY 2: IMPOSE STRUCTURE

Heap Sort

WORST $\theta(n \log n)$

BEST $\theta(n)$

Always good runtimes, great if already sorted.

IN-PLACE SPACE $\theta(1)$

STRATEGY 3: DIVIDE AND CONQUER

Merge Sort

WORST $\theta(n \log n)$

BEST $\theta(n \log n)$

Stable, very reliable! In-place variant is slower.

STABLE SPACE $\theta(n)$

Quick Sort

WORST $\theta(n^2)$

BEST $\theta(n \log n)$

Fastest in practice (constant factors), bad worst case.

IN-PLACE SPACE $\theta(1)$

Insertion Sort

WORST $\theta(n^2)$

BEST $\theta(n)$

Simple, stable, low-overhead, great if already sorted.

IN-PLACE STABLE SPACE $\theta(1)$

Heap Sort

WORST $\theta(n \log n)$

BEST $\theta(n)$

Always good runtimes, great if already sorted.

IN-PLACE SPACE $\theta(1)$

Merge Sort

WORST $\theta(n \log n)$

BEST $\theta(n \log n)$

Stable, very reliable! In-place variant is slower.

STABLE SPACE $\theta(n)$

Selection Sort

WORST $\theta(n^2)$

BEST $\theta(n^2)$

Minimizes array writes, otherwise never preferred.

IN-PLACE SPACE $\theta(1)$

Quick Sort

WORST $\theta(n^2)$

BEST $\theta(n \log n)$

Fastest in practice (constant factors), bad worst case.

IN-PLACE SPACE $\theta(1)$

Can we do better than n log n?

- For comparison sorts, **NO**. A proven upper bound!
  - Intuition: n elements to sort, no faster way to find “right place” than log n
- However, niche sorts can do better in specific situations!

Many cool niche sorts beyond the scope of 373!

Radix Sort (Wikipedia, VisuAlgo) - Go digit-by-digit in integer data. Only 10 digits, so no need to compare!

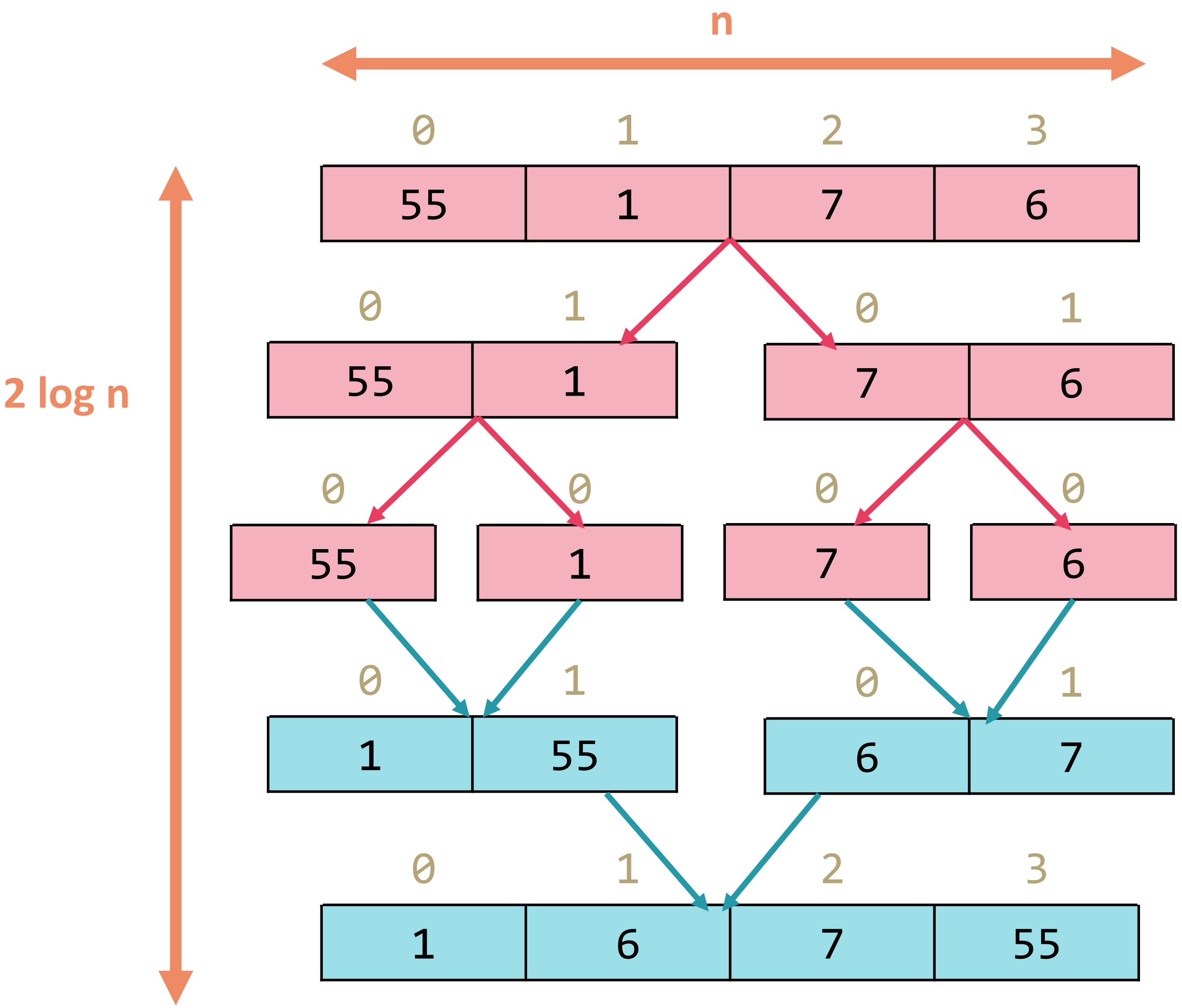
Counting Sort (Wikipedia)

Bucket Sort (Wikipedia)

External Sorting Algorithms (Wikipedia) - For big data™

Review Merge Sort

```
mergeSort(list) {
   if (list.length == 1):
      return list
   else:
      smallerHalf = mergeSort(new [0, ..., mid])
      largerHalf = mergeSort(new [mid + 1, ...])
      return merge(smallerHalf, largerHalf)
}
```



Worst case runtime? $T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + n & \text{otherwise} \end{cases}$

$= \Theta(n \log n)$

Best case runtime? Same

In Practice runtime? Same

Stable? Yes

In-place? No

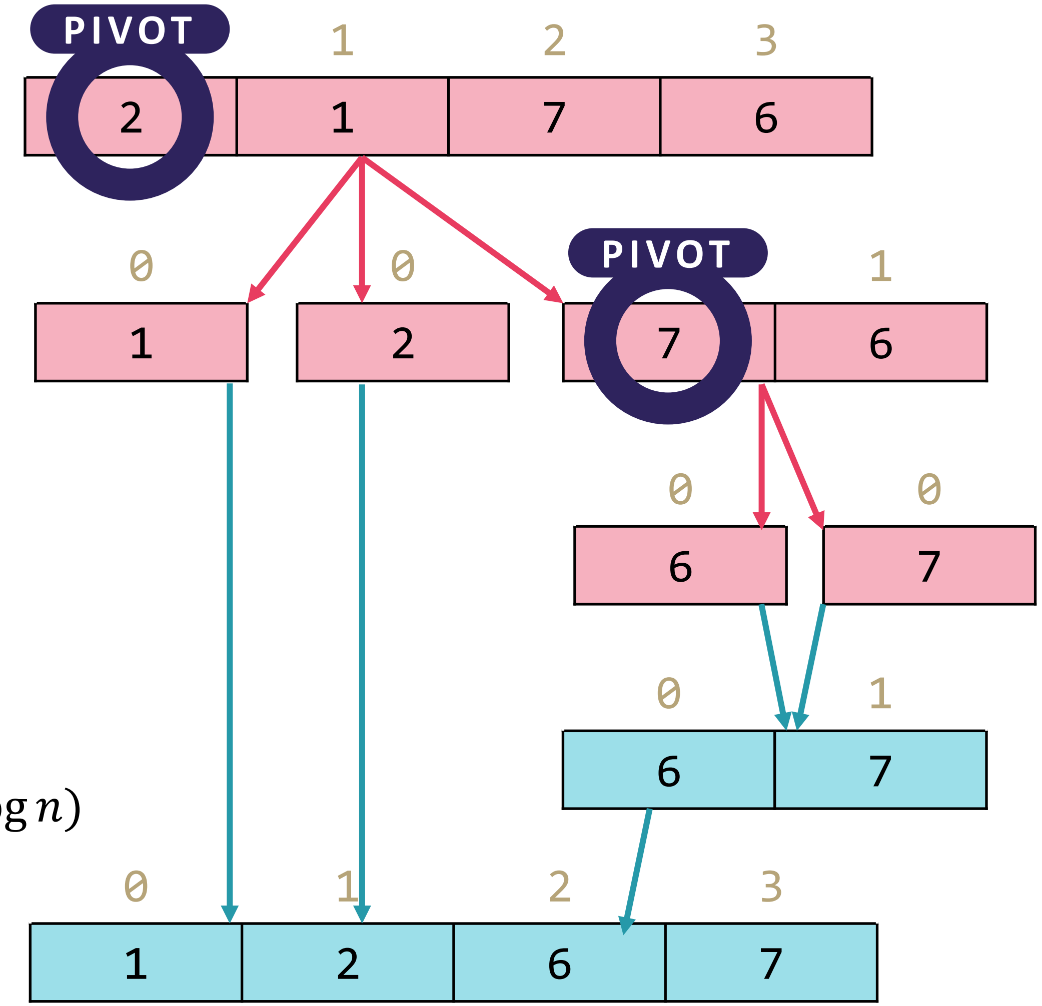
2 **Constant size Input**

Don't forget your old friends, the 3 recursive patterns!

Review Quick Sort (v1)

Worst case: Pivot only chops off one value
Best case: Pivot divides each array in half

```
quickSort(list) {
   if (list.length == 1):
      return list
   else:
      pivot = choosePivot(list)
      smallerHalf = quickSort(getSmaller(pivot, list))
      largerHalf = quickSort(getBigger(pivot, list))
      return smallerHalf + pivot + largerHalf
}
```



Worst case runtime? $T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ T(n-1) + n & \text{otherwise} \end{cases} = \Theta(n^2)$

Best case runtime? $T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + n & \text{otherwise} \end{cases} = \Theta(n \log n)$

In-practice runtime? Just trust me: $\Theta(n \log n)$
(absurd amount of math to get here)

Stable? No

In-place? Can be done!

Review Strategies for Choosing a Pivot

- Just take the first element
  - Very fast!
  - But has worst case: for example, sorted lists have $\Omega(n^2)$ behavior
- Take the median of the first, last, and middle element
  - Makes pivot slightly more content-aware, at least won't select very smallest/largest
  - Worst case is still $\Omega(n^2)$, but on real-world data tends to perform well!

Most commonly used

- Take the median of the full array
  - Can actually find the median in $O(n)$ time (google QuickSelect). It's **complicated.**
  - $O(n \, log \, n)$ even in the worst case… but the constant factors are **awful**. No one does quicksort this way.
- Pick a random element
  - Get $O(n \log n)$ runtime with probability at least $1 - 1/n^2$
  - No simple worst-case input (e.g. sorted, reverse sorted)

Lecture Outline

- **Comparison Sorts**
  - *Review* Sorting Overview
  - **In-Place Quick Sort**
- Topological Sort
- Reductions
  - Definitions
  - Examples

Quick Sort (v2: In-Place)

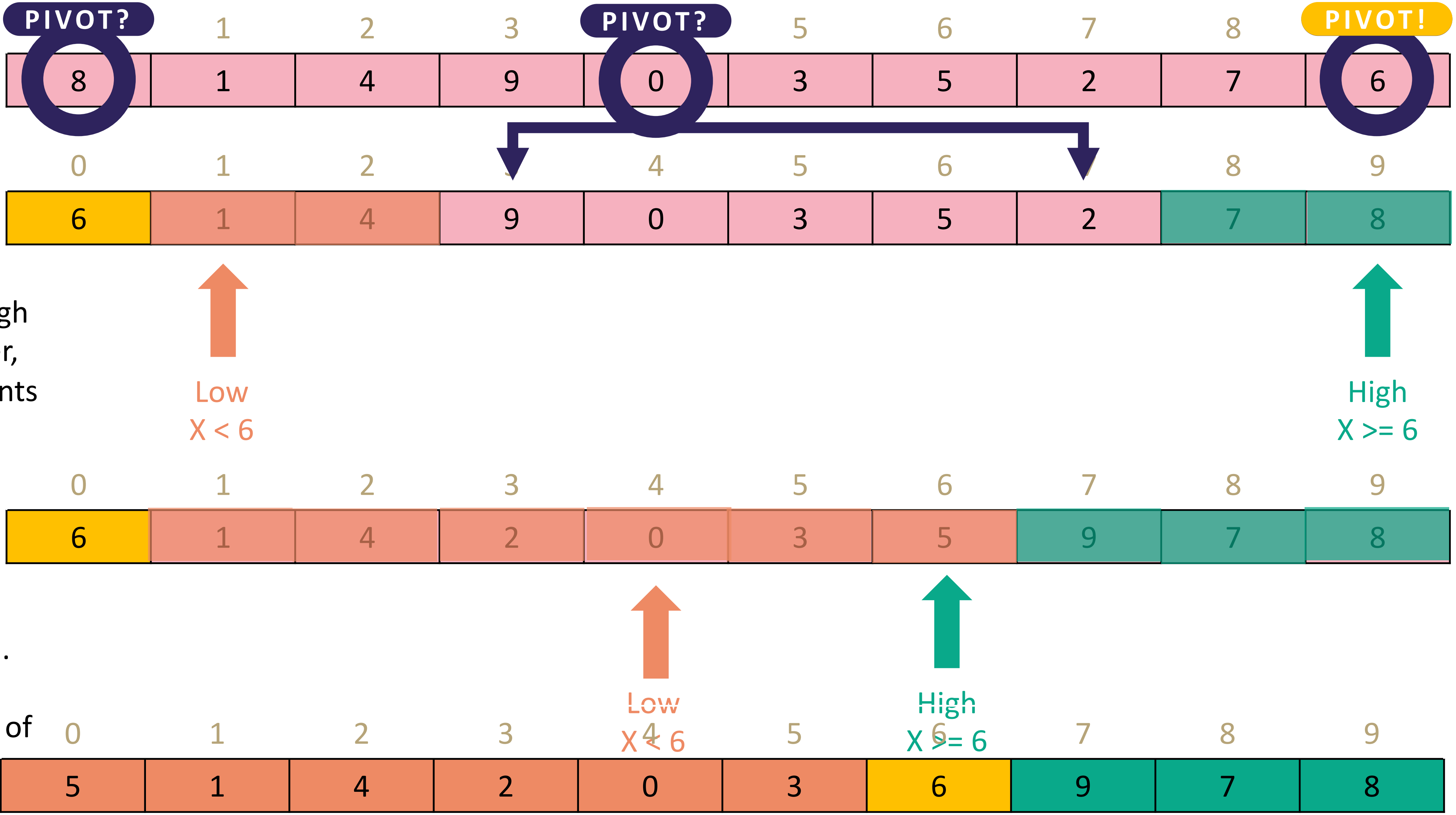
Divide

Select a pivot

Move pivot out of the way

Bring low and high pointers together, swapping elements if needed

Meeting point is where pivot belongs; swap in. Now recurse on smaller portions of same array!



Quick Sort (v2: In-Place)

```
quickSort(list) {
   if (list.length == 1):
      return list
   else:
      pivot = choosePivot(list)
      smallerPart, largerPart = partition(pivot, list)
      smallerPart = quickSort(smallerPart)
      largerPart = quickSort(largerPart)
      return smallerPart + pivot + largerPart
}
```

choosePivot:
- Use one of the pivot selection strategies

partition:
- For in-place Quick Sort, series of swaps to build both partitions at once
- Tricky part: moving pivot out of the way and moving it back!
- Similar to Merge Sort divide step: two pointers, only move smaller one

Worst case runtime? $T(n) = \begin{cases} 1 & \text{if } n \le 1 \\ T(n-1) + n & \text{otherwise} \end{cases} = \Theta(n^2)$

Best case runtime? $T(n) = \begin{cases} 1 & \text{if } n \le 1 \\ 2T\left(\frac{n}{2}\right) + n & \text{otherwise} \end{cases} = \Theta(n \log n)$

In-practice runtime? Just trust me: $\Theta(n \log n)$
(absurd amount of math to get here)

Stable? No

In-place? Yes

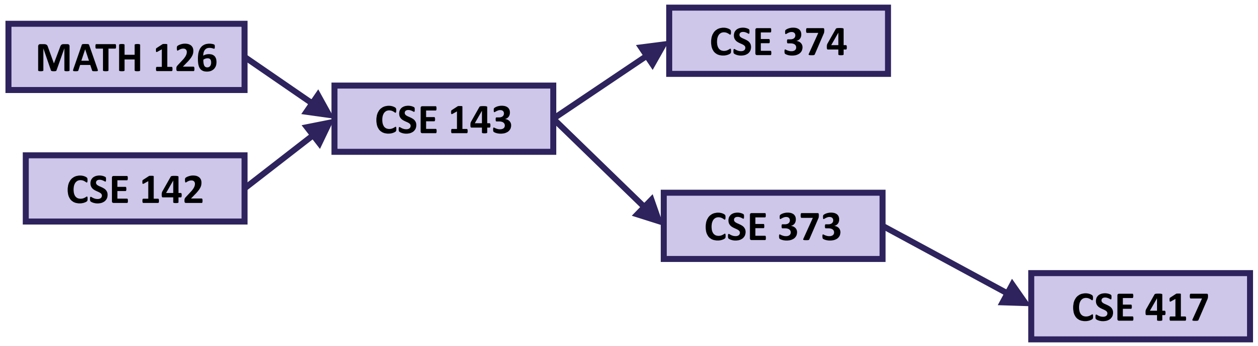
0	1	2	3	4	5
0	3	6	9	7	8

Lecture Outline

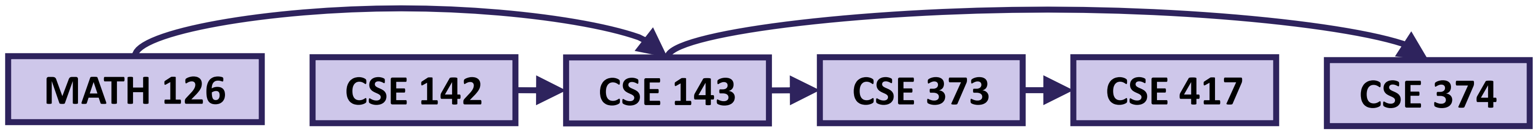
- Comparison Sorts
  - *Review* Sorting Overview
  - In-Place Quick Sort
- **Topological Sort**
- Reductions
  - Definitions
  - Examples

Sorting Dependencies

- Given a set of courses and their prerequisites, find an order to take the courses in (assuming you can only take one course per quarter)



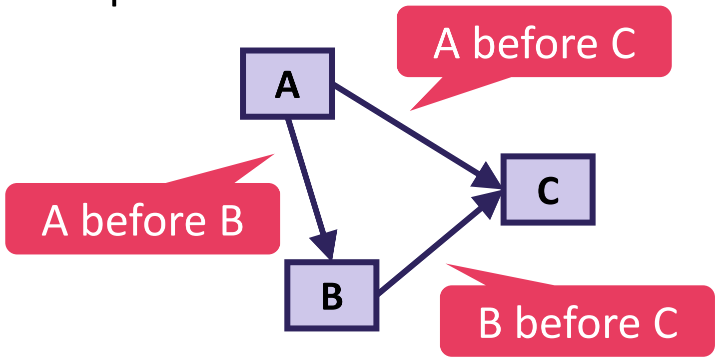
- Possible ordering:



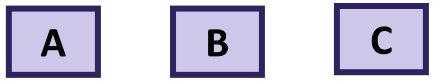
Topological Sort

- A **topological sort** of a directed graph G is one where for every edge, the origin appears before the destination
- 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

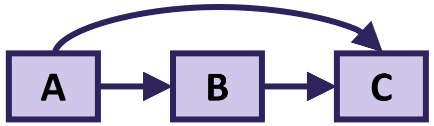
Input:



Topological Sort:

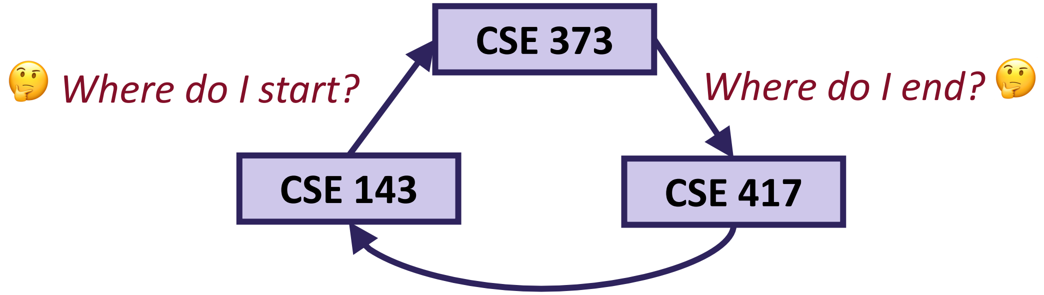


With original edges for reference:



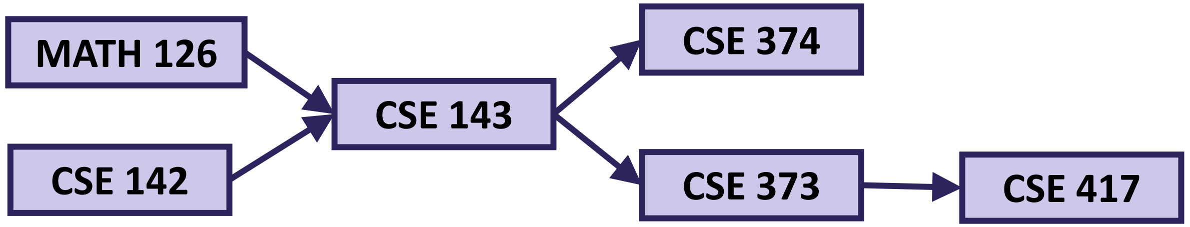
Can We Always Topo Sort a Graph?

- Can you topologically sort this graph?



No 😭

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



- A graph has a topological ordering iff 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

How To Perform Topo Sort?

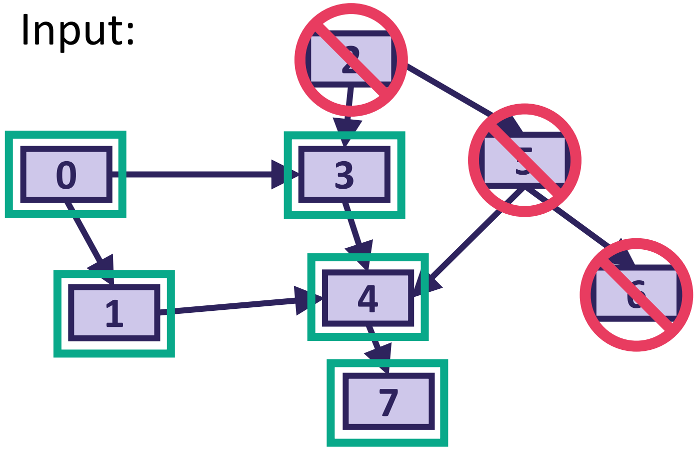
- Topo sort is an ordering problem. Could we use… BFS?

IDEA 1 **Performing Topo Sort**

Use BFS, starting from a vertex with no incoming edges

Doesn't reach all vertices ☹

Input:



BFS starting at 0:

How To Perform Topo Sort?

- Okay, there may be multiple “roots”. What if we use BFS multiple times?

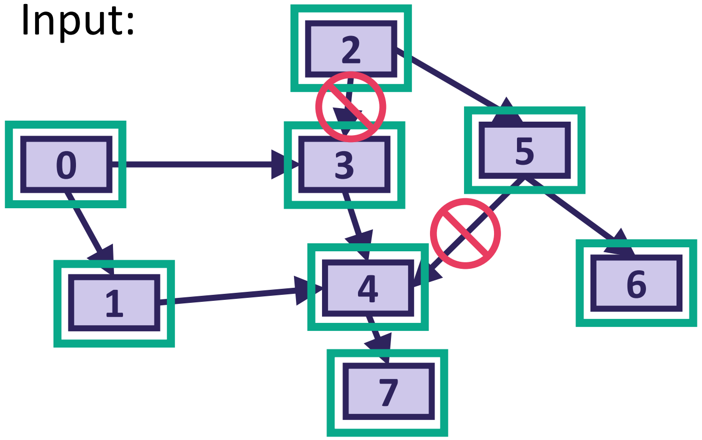
IDEA 2

Performing Topo Sort

Use BFS, starting from ALL vertices with no incoming edges

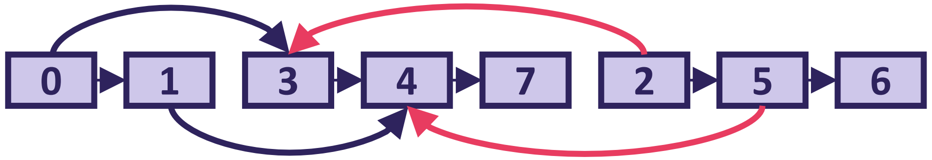
Doesn’t respect all edges ☹

Input:



BFS starting at 0:

+ BFS starting at 2:



CSE 373

- Home
- Projects
- Exercises
- Exams
- Office Hours
- Staff
- Syllabus

Course Tools

- Zoom Information
- Piazza
- Gradescope
- GitLab
- Lecture Recordings
- Anonymous Feedback

Acknowledgements

Resources: review videos

Fri 07/31 LEC 16 Dijkstra's Algorithm
Slides: pdf pptx
Resources: video, optional review, (original video)

Week 7

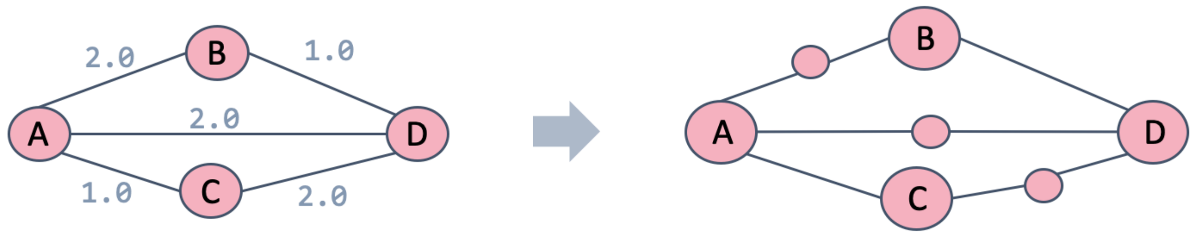
P3 Heap

RELEASED

Idea 1: Change into an unweighted graph

- We know BFS works on unweighted graphs
  - If we can transform a weighted graph to unweighted, we can solve it!

- This idea is known as a **reduction**
  - "Reduce" a problem you can't solve to one you can
  - Here, we're trying to reduce BFS on weighted graphs to BFS on unweighted graphs
  - We'll revisit this concept later in the course!



Fri 08/14 LEC 22 Topo Sort & Reductions

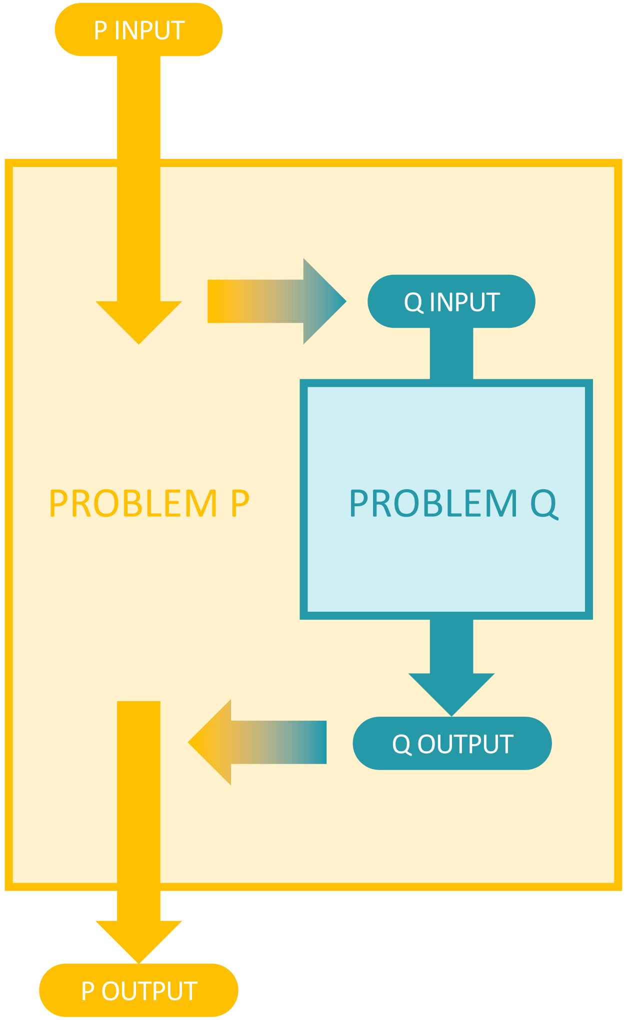
Lecture Outline

- Comparison Sorts
  - *Review* Sorting Overview
  - In-Place Quick Sort
- Topological Sort
- **Reductions**
  - **Definitions**
  - Examples

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"

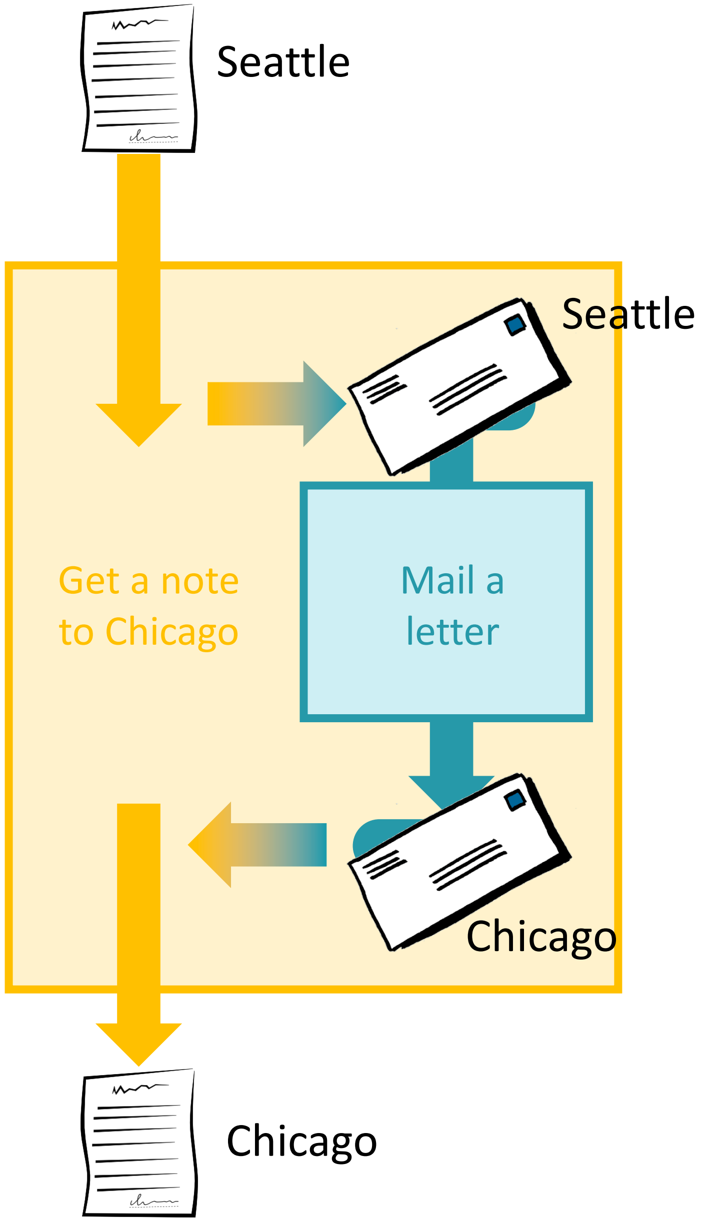
1. Convert input for P into input for Q
2. Solve using algorithm for Q
3. Convert output from Q into output from P



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?

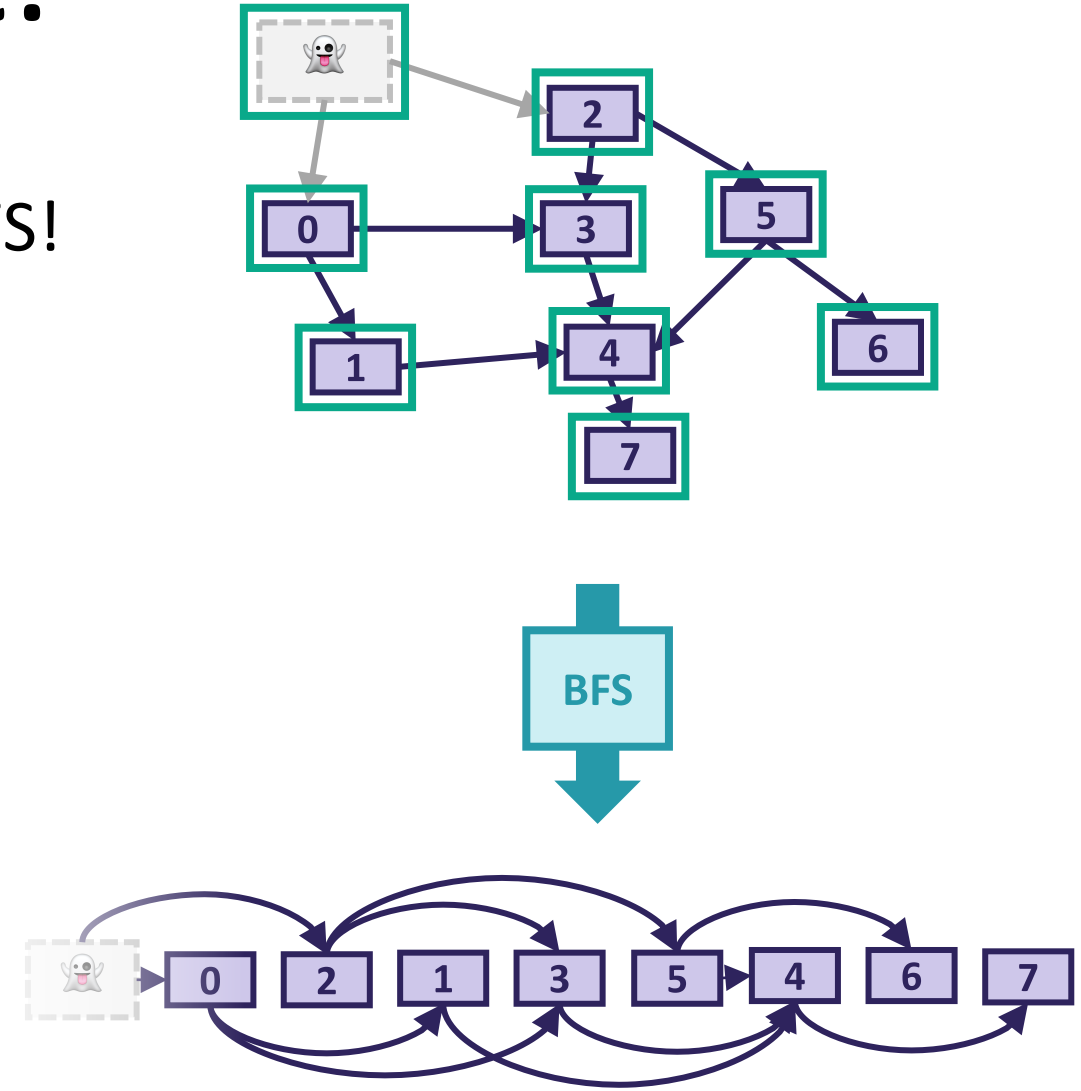
- If we add a phantom "start" vertex pointing to other starts, we could use BFS!

IDEA 3

Performing Topo Sort

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

Sweet sweet victory 😎



Poll Everywhere pollev.com/uwcse373

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

Are Prim's and Dijkstra's related via a reduction?

a) Yes.
   Prim's reduces to Dijkstra's.

b) Yes.
   Dijkstra's reduces to Prim's.

c) No.
   This is not a reduction.

In a reduction, we modify inputs/outputs, not the algorithm itself!

Lecture Outline

- Comparison Sorts
  - *Review* Sorting Overview
  - In-Place Quick Sort
- Topological Sort
- **Reductions**
  - Definitions
  - **Examples**

Checking for Duplicates

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

- Initial idea: Compare every element to every other element!
  - Runtime: $\theta(n^2)$

0	1	2	3	4
2	4	8	3	8

```
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
}
```

- Could we do better?

Poll Everywhere pollev.com/uwcse373

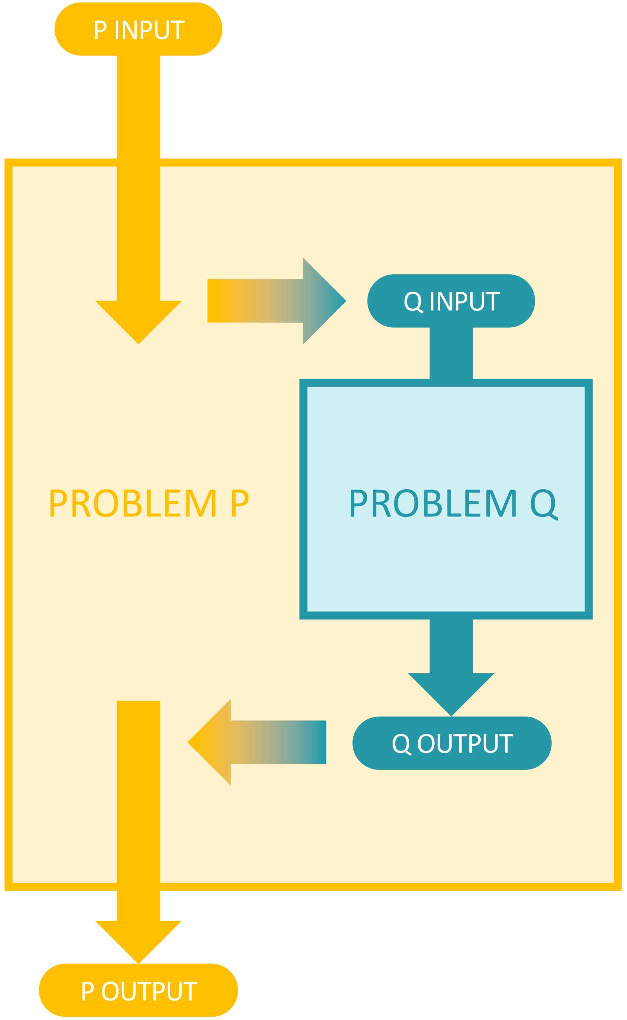
Your Turn!

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



Poll Everywhere

pollev.com/uwcse373

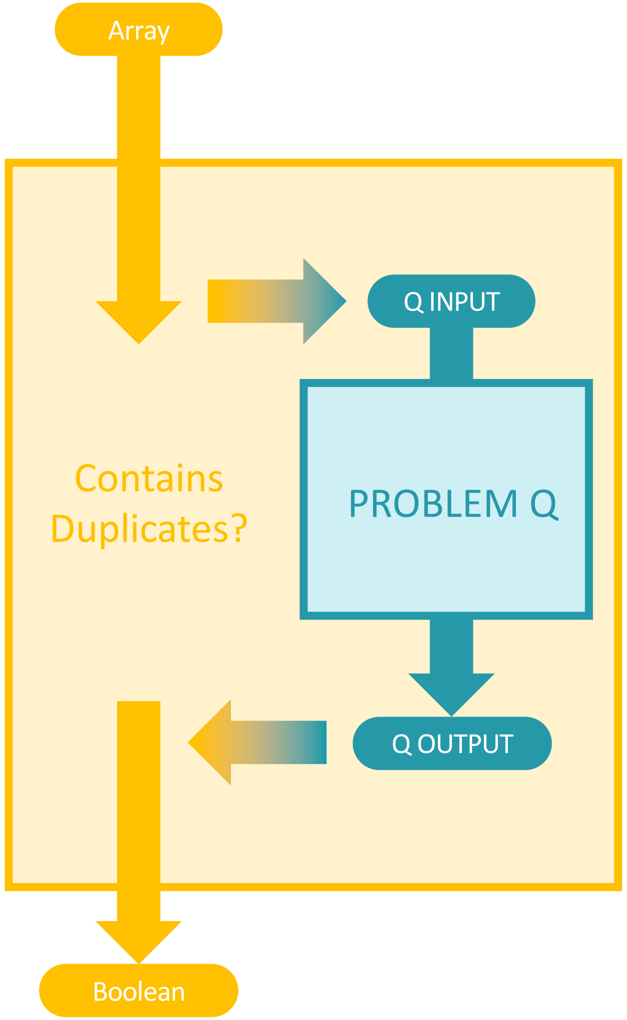
Your Turn!

How would you reduce "Contains Duplicates?" to another problem?

Top

Total Results: 0

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**



Poll Everywhere pollev.com/uwcse373

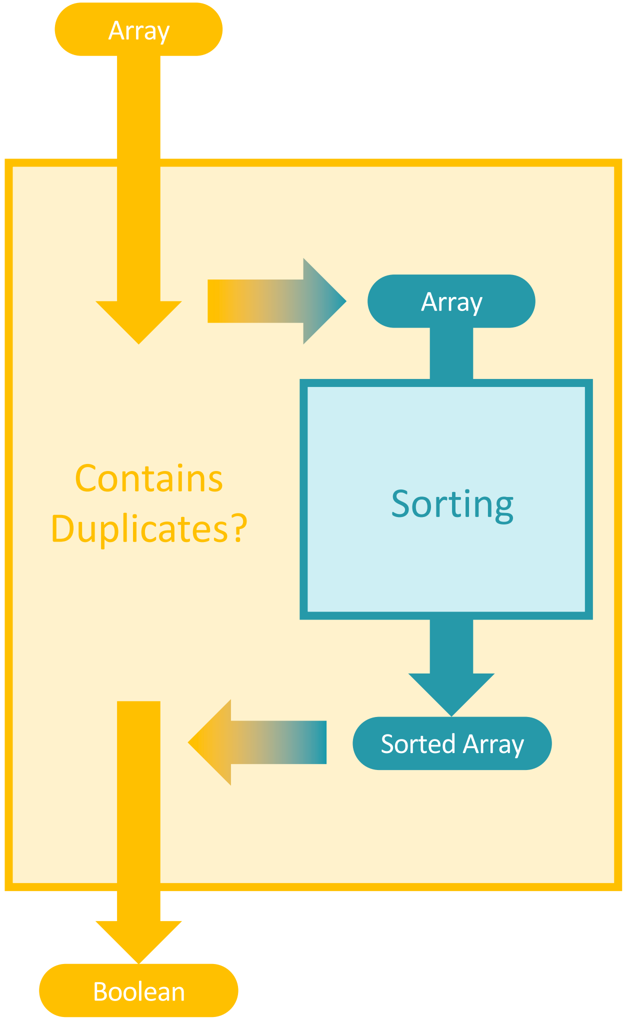
Your Turn!

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!



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

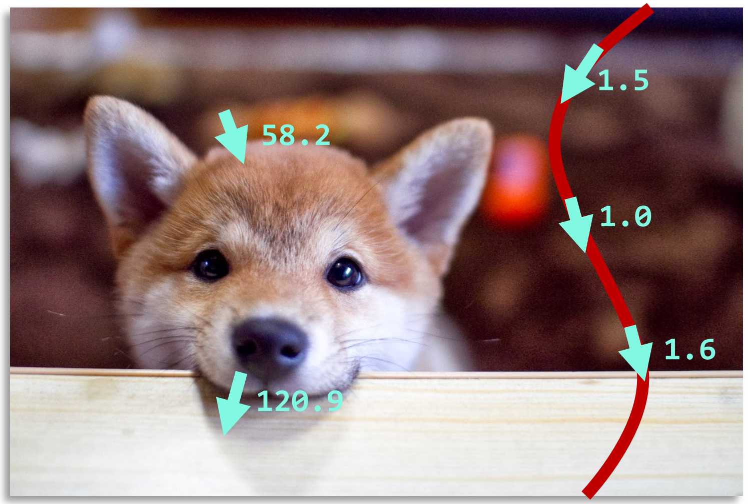
Seam Carving Reduces to Dijkstra's!

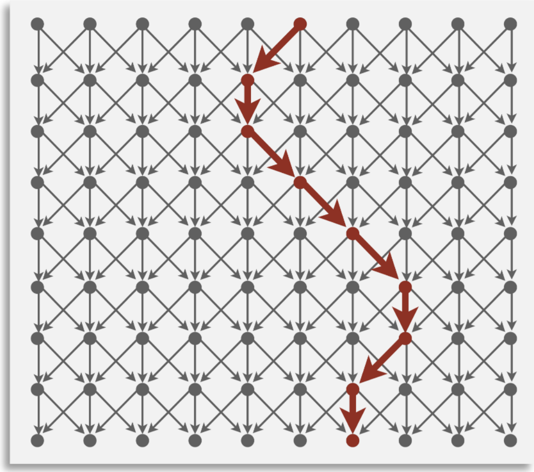
1. *Transform the input so that it can be solved by the standard algorithm*
   - Formulate the image as a graph
     - **Vertices**: pixel in the image
     - **Edges**: connects a pixel to its 3 downward neighbors
     - **Edge Weights**: the "energy" (**visual difference**) between adjacent pixels

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

greater pixel difference = higher weight!



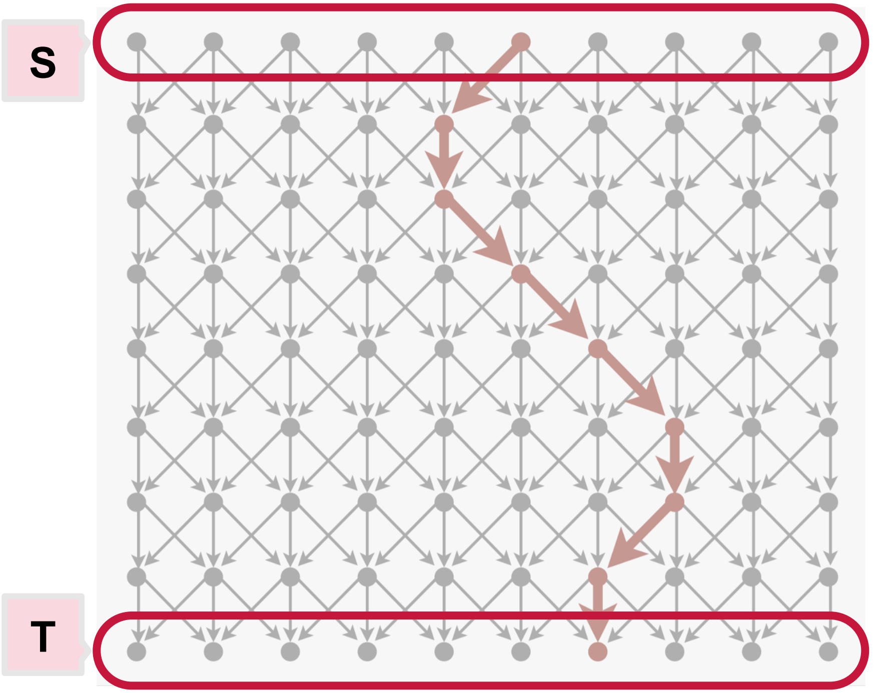


Shortest Paths (Robert Sedgewick, Kevin Wayne/Princeton)

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?



Shortest Paths (Robert Sedgewick, Kevin Wayne/Princeton)

In Conclusion

- Topo Sort is a widely applicable "sorting" algorithm beyond the classic comparison sorts
- 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 week 9 in this course *reduces to* starting early on P4 and EX4