CSE 373: Data Structures and Algorithms

# Graphs

Autumn 2018

Shrirang (Shri) Mare
shri@cs.washington.edu

Thanks to Kasey Champion, Ben Jones, Adam Blank, Michael Lee, Evan McCarty, Robbie Weber, Whitaker Brand, Zora Fung, Stuart Reges, Justin Hsia, Ruth Anderson, and many others for sample slides and materials ...

# Technique 3: Master Theorem

Review

Given a recurrence of the following form:

$$T(n) = \begin{cases} d & when\ n = 1 \\ aT\left(\frac{n}{b}\right) + n^c & otherwise \end{cases}$$

Where $a$, $b$, and $c$ are constants, then $T(n)$ has the following asymptotic bounds

If $\log_b a < c$ then $T(n) \in \Theta(n^c)$

If $\log_b a = c$ then $T(n) \in \Theta(n^c \log_2 n)$

If $\log_b a > c$ then $T(n) \in \Theta\left(n^{\log_b a}\right)$

# Recurrence analysis techniques

Review

1. Unfolding method
   - more of a brute force method
   - Tedious but works
2. Tree methods
   - more scratch work but less error prone
3. Master theorem
   - quick, but applicable only to certain type of recurrences
   - does not give a closed form (gives big-Theta)

# Desired properties in a sorting algorithm

Review

Stable

- In the output, equal elements (i.e., elements with equal keys) appear in their original order

In-place

- Algorithm uses a constant additional space, $O(1)$ extra space

Adaptive

- Performs better when input is almost sorted or nearly sorted
- (Likely different big-O for best-case and worst-case)

Fast. $O\ (n \log n)$

No algorithm has all of these properties. So choice of algorithm depends on the situation.

# Merge sort

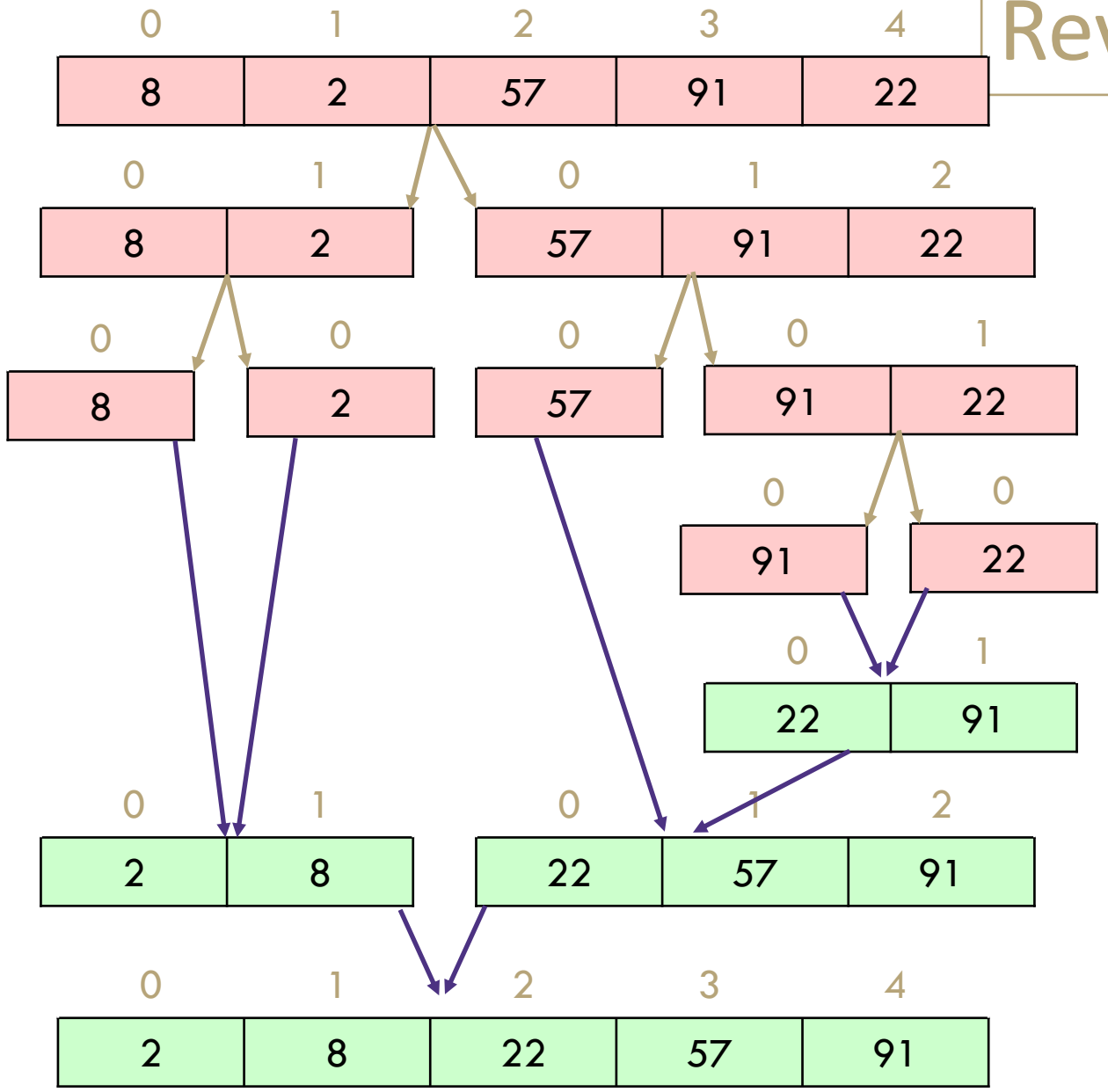
Review

Split array in the middle

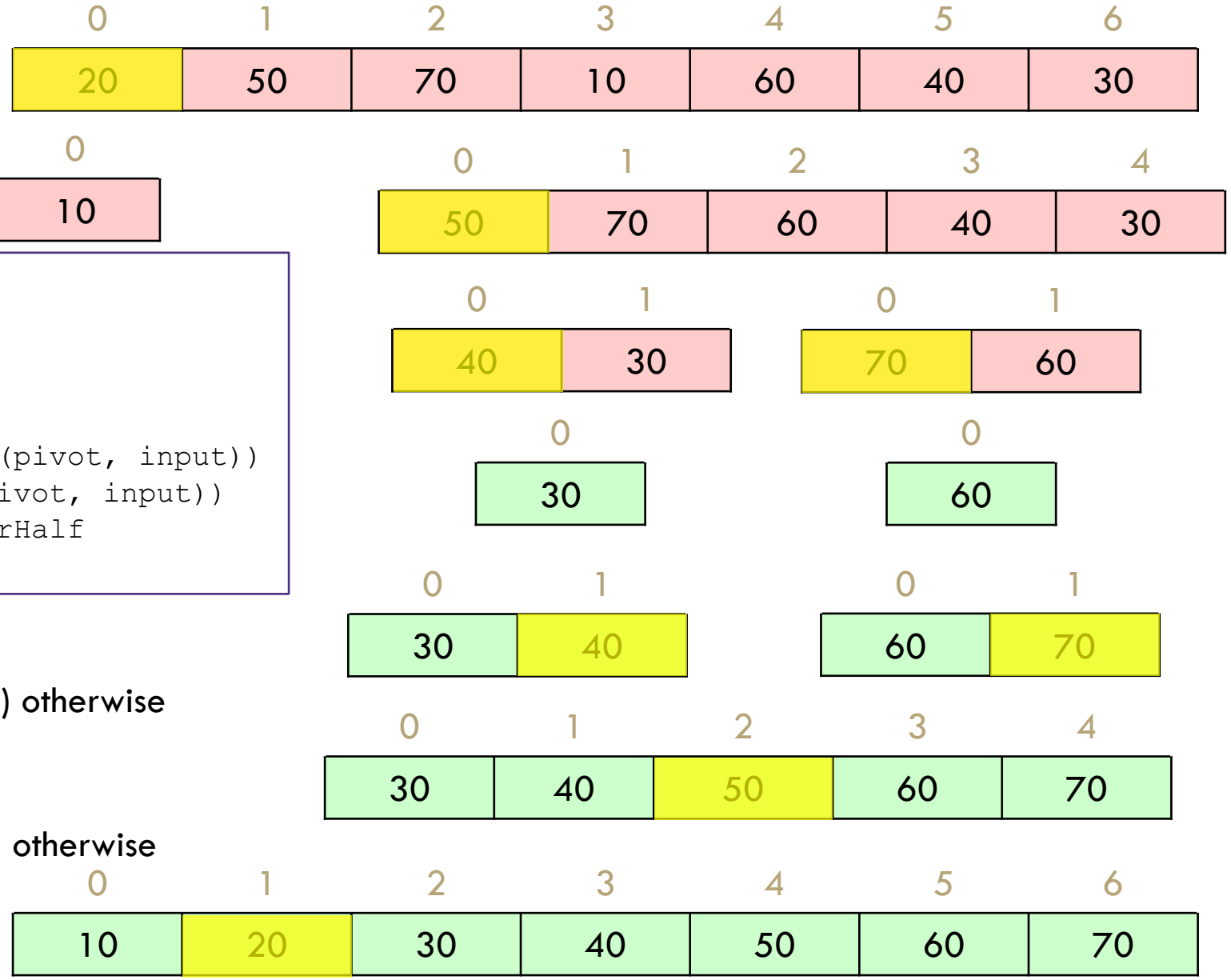
Sort the two halves

Merge them together

$$T(n) = \begin{cases} c_1 & \text{if } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + c_2 n & \text{otherwise} \end{cases}$$



# Quick Sort




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

Worst case runtime? $T(n) = \begin{cases} 1 \text{ if } n <= 1 \\ n + T(n - 1) \text{ otherwise} \end{cases}$

Best case runtime?

Average runtime? $T(n) = \begin{cases} 1 \text{ if } n <= 1 \\ n + 2T(n/2) \text{ otherwise} \end{cases}$

Stable? No

In-place? Can be

# Choosing a Pivot

Average case behavior depends on a good pivot.

Pivot ideas:

Just take the first element

- Simple. But an already sorted (or reversed) list will give you a bad time.

Pick an element uniformly at random.

- Regardless of input!
- Probably too slow in practice :(

Median of Three

- Take the median of the first, last, and midpoint as the pivot.
- Fast!
- Unlikely to get bad behavior (but definitely still possible)
- Reasonable default choice.

# Worksheet Questions 1-3

# Parting thoughts on sorts

- Hybrid sorts
- Internal vs. external sorting

# Graphs

# Inter-data Relationships

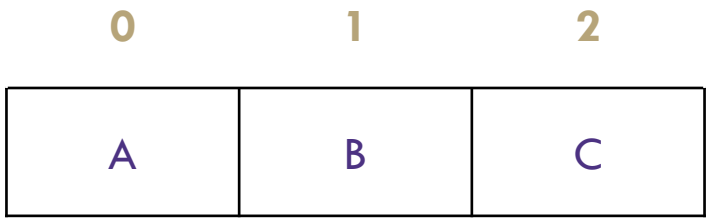
## Arrays

Categorically associated

Sometimes ordered

Typically independent

Elements only store pure data, no connection info



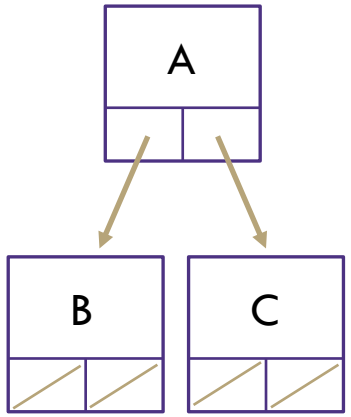

## Trees

Directional Relationships

Ordered for easy access

Limited connections

Elements store data and connection info



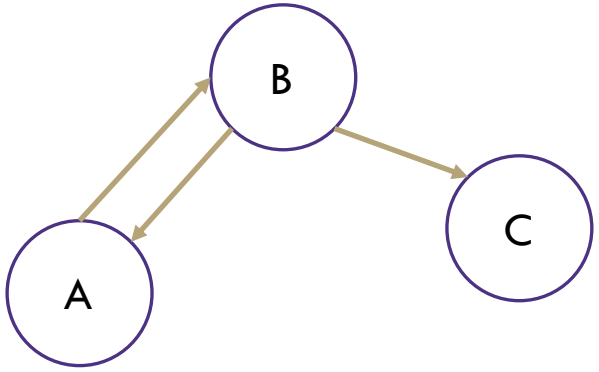

## Graphs

Multiple relationship connections

Relationships dictate structure

Connection freedom!

Both elements and connections can store data



# Applications

## Physical Maps
- Airline maps
  - Vertices are airports, edges are flight paths
- Traffic
  - Vertices are addresses, edges are streets

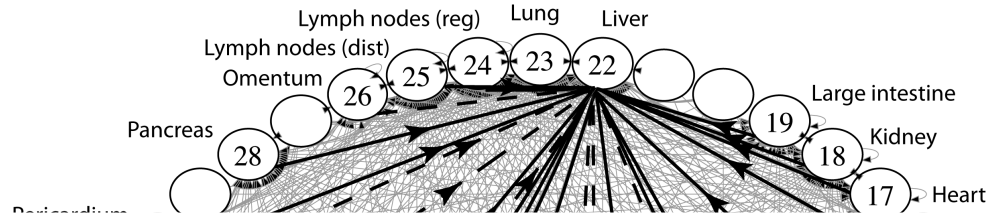
## Relationships
- Social media graphs
  - Vertices are accounts, edges are follower relationships
- Code bases
  - Vertices are classes, edges are usage

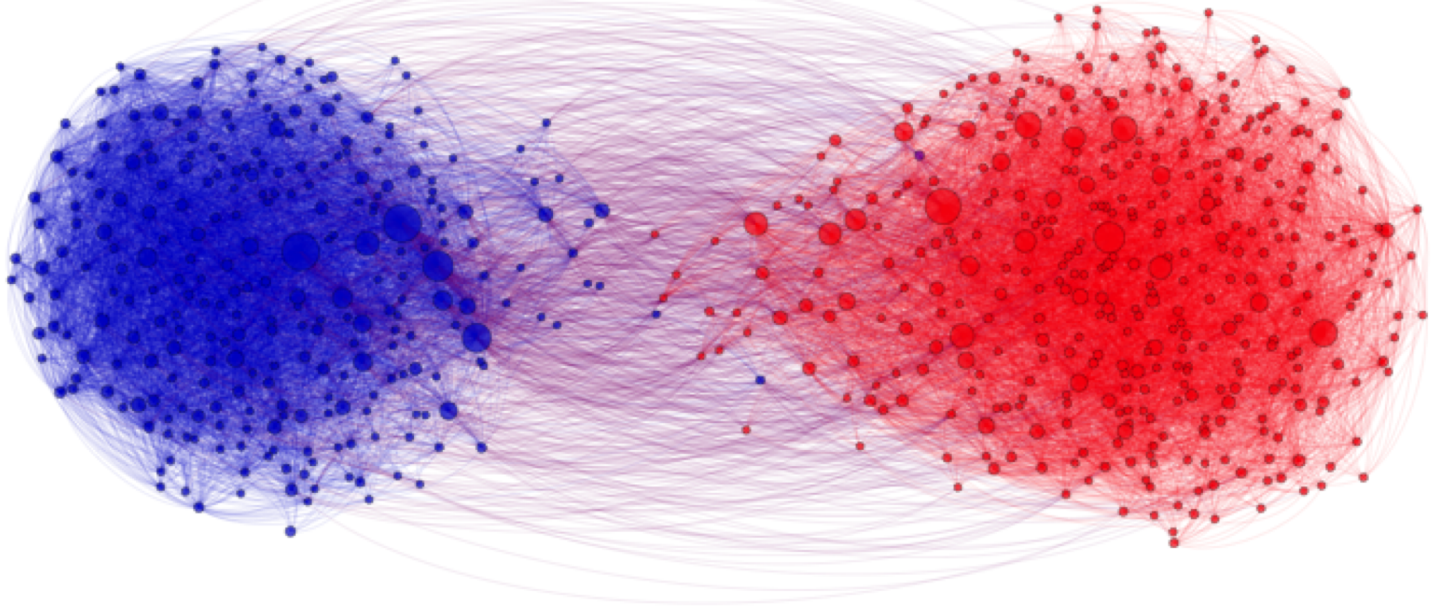
## Influence
- Biology
  - Vertices are cancer cell destinations, edges are migration paths

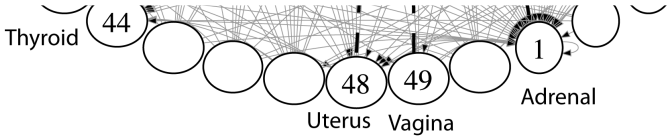
## Related topics
- Web Page Ranking
  - Vertices are web pages, edges are hyperlinks
- Wikipedia
  - Vertices are articles, edges are links

SO MANY MORREEEE

[www.allthingsgraphed.com](http://www.allthingsgraphed.com)





# Graph Vocabulary

## Graph Direction

- **Undirected graph** – edges have no direction and are two-way

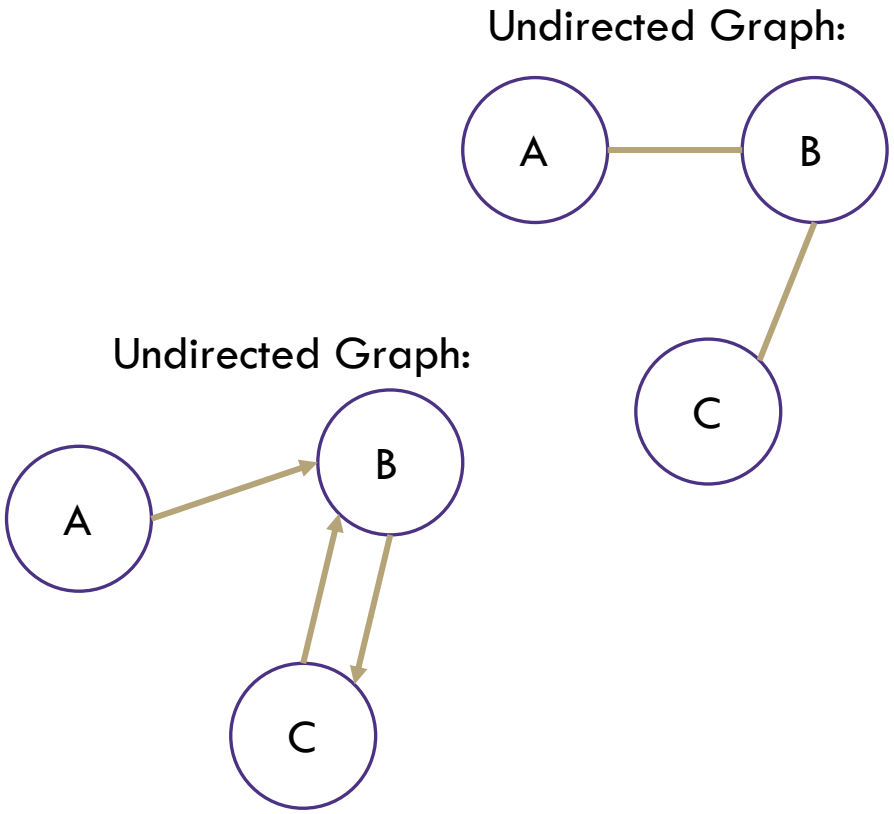
  V = { A, B, C }

  E = { (A, B), (B, C) } *inferred (B, A) and (C,B)*

- **Directed graphs** – edges have direction and are thus one-way

  V = { A, B, C }

  E = { (A, B), (B, C), (C, B) }




## Degree of a Vertex

- **Degree** – the number of edges containing that vertex

  A : 1, B : 1, C : 1

- **In-degree** – the number of directed edges that point to a vertex

  A : 0, B : 2, C : 1

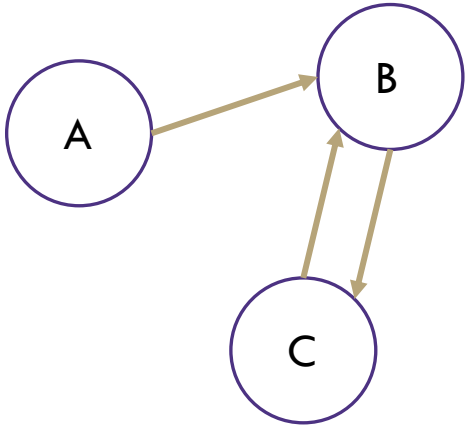
- **Out-degree** – the number of directed edges that start at a vertex

  A : 1, B : 1, C : 1

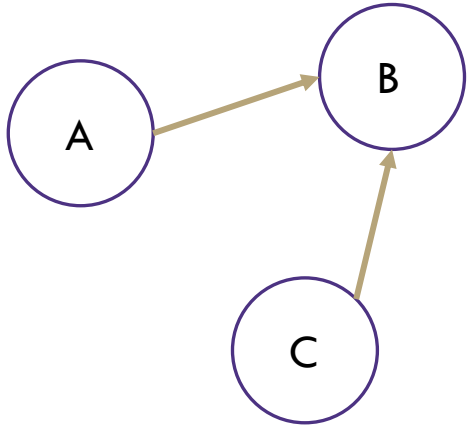
# Food for thought

Is a graph valid if there exists a vertex with a degree of 0?

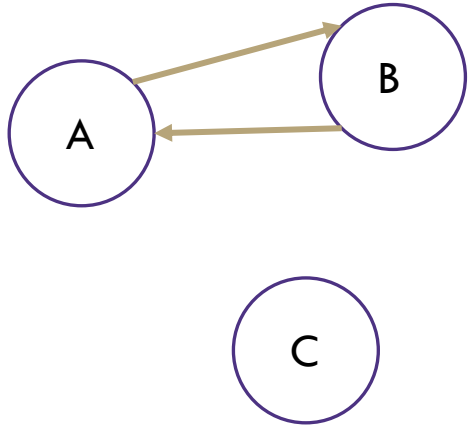
Yes




A has an "in degree" of 0

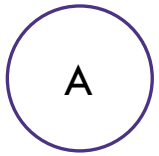



B has an "out degree" of 0




C has both an "in degree" and an "out degree" of 0

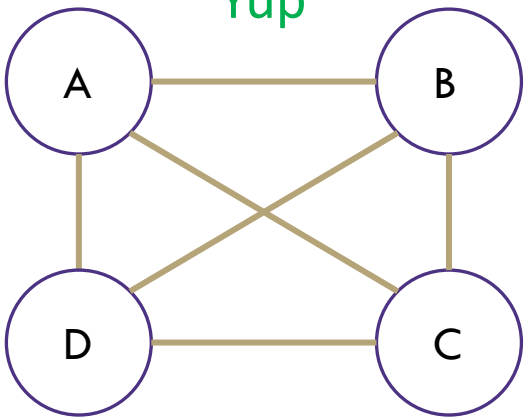
Is this a valid graph?

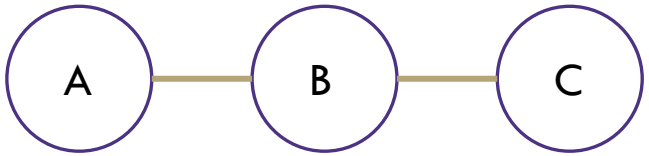



Yes!

Are these valid?

Yup

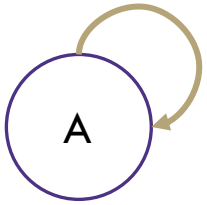


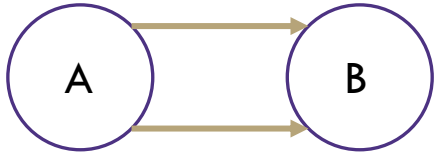

Sure

# Graph Vocabulary

**Self loop** – an edge that starts and ends at the same vertex



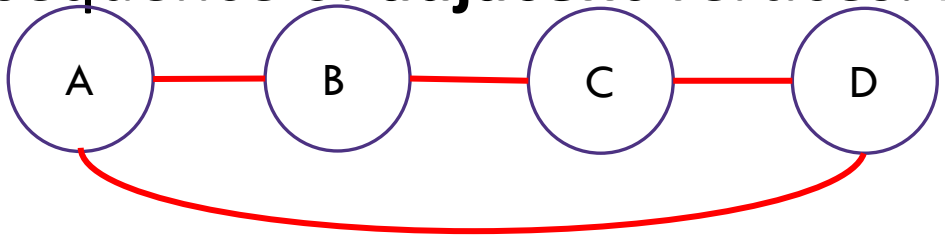

**Parallel edges** – two edges with the same start and end vertices




**Simple graph** – a graph with no self-loops and no parallel edges

# Graph Vocabulary

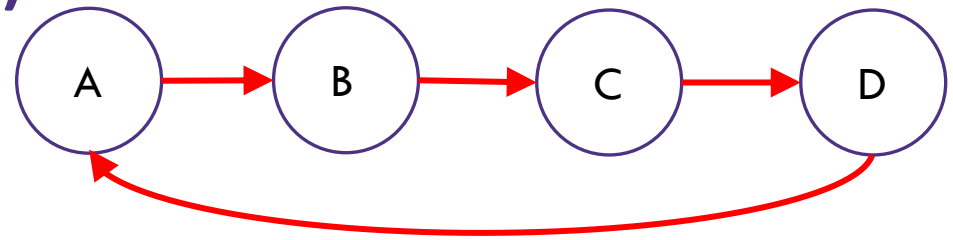
**Walk** – A sequence of **adjacent** vertices. Each connected to next by an edge.




A,B,C,D is a walk.

So is A,B,A

**(Directed) Walk**–must follow the direction of the edges




A,B,C,D,B is a directed walk.

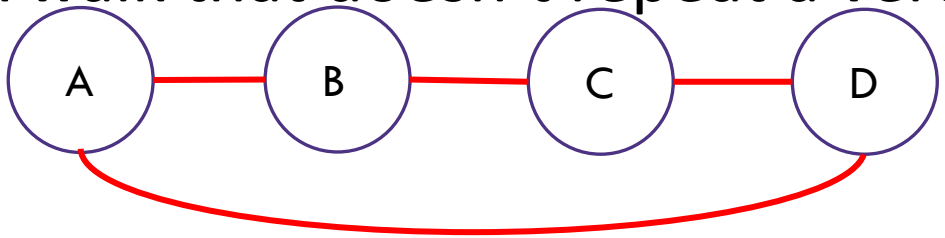
A,B,A is not.

**Length** – The number of edges in a walk

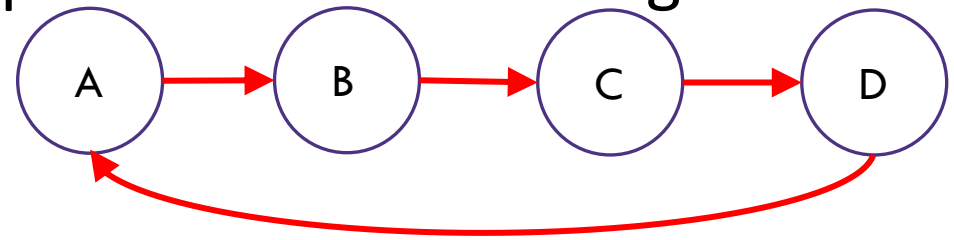
- (A,B,C,D) has length 3.

# Graph Vocabulary

**Path** – A walk that doesn't repeat a vertex. A,B,C,D is a path. A,B,A is not.




**Cycle** – path with an extra edge from last vertex back to first.




Be careful looking at other sources.

Some people call our "walks" "paths" and our "paths" "simple paths"

Use the definitions on these slides.

# Worksheet Question 4

# Implementing a Graph

Implement with nodes…

Implementation gets super messy

What if you wanted a vertex without an edge?

How can we implement without requiring edges to access nodes?

Implement using some of our existing data structures!

# Adjacency Matrix

Assign each vertex a number from 0 to V – 1

Create a V x V array of Booleans

If (x,y) ∈ E then arr[x][y] = true

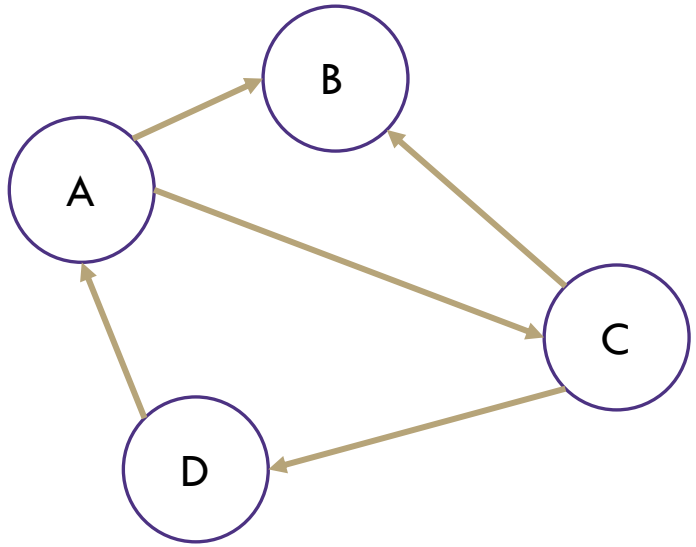
| | A | B | C | D |
|---|---|---|---|---|
| A | | T | T | |
| B | | | | |
| C | | T | | T |
| D | T | | | |

Runtime (in terms of V and E)

- get out - edges for a vertex O(v)
- get in – edges for a vertex O(v)
- decide if an edge exists O(1)
- insert an edge O(1)
- delete an edge O(1)
- delete a vertex
- add a vertex

How much space is used?

$V^2$



# Graph Vocabulary

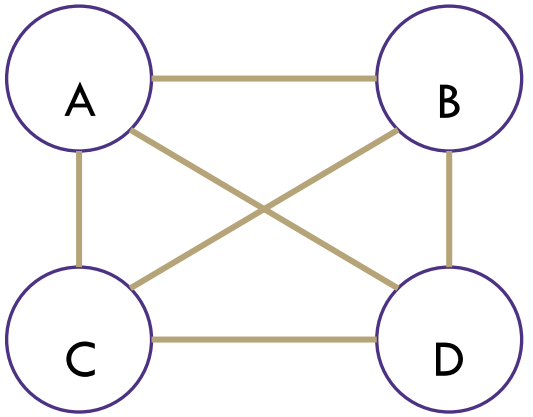
**Dense Graph** – a graph with a lot of edges

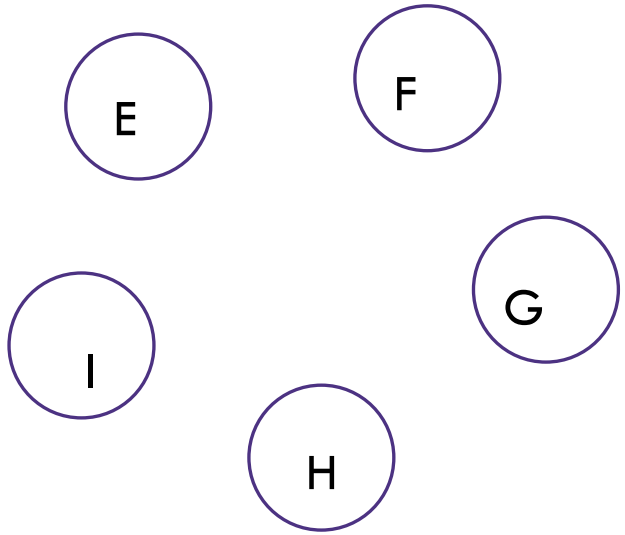
$E \in \Theta(V^2)$

**Sparse Graph** – a graph with "few" edges

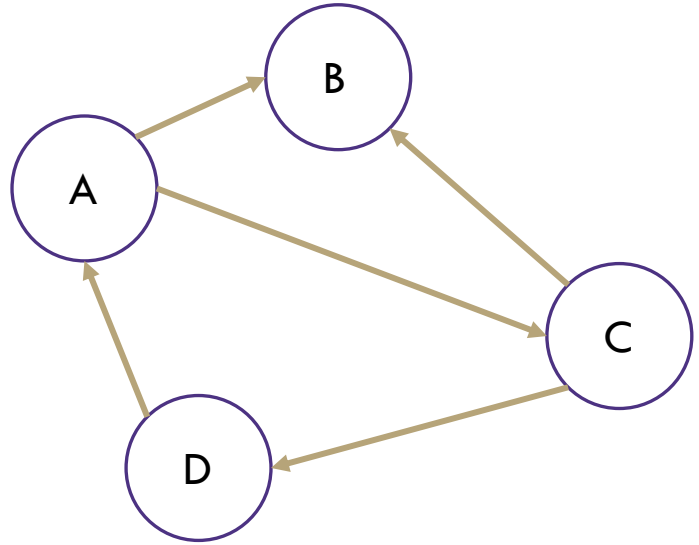
$E \in \Theta(V)$

An Adjacency Matrix seems a waste for a sparse graph…



# Adjacency List




Create a Dictionary of size V from type V to Collection of E

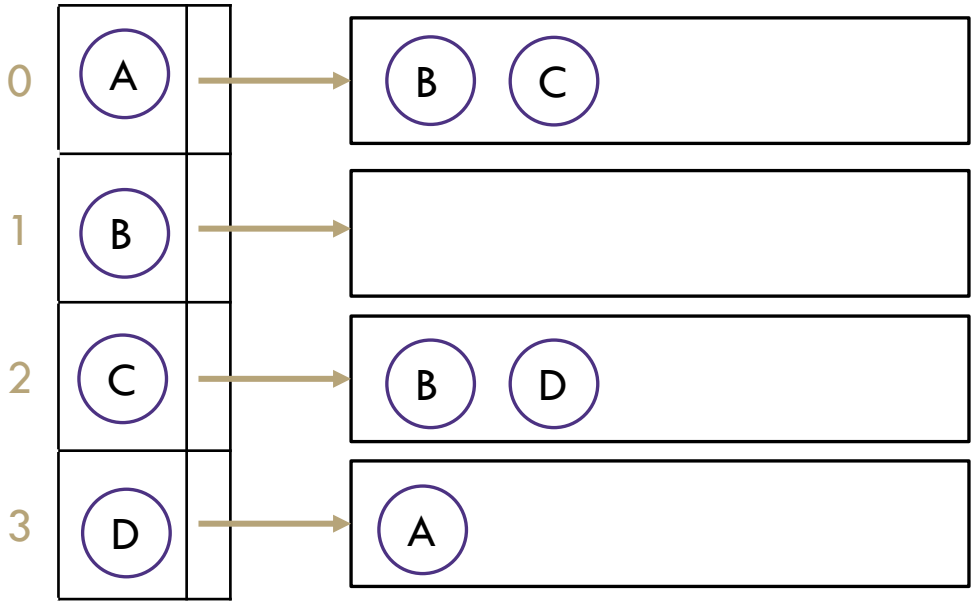
If (x,y) ∈ E then add y to the set associated with the key x

Runtime (in terms of V and E)

- get out - edges for a vertex O(1)
- get in - edges for a vertex O(V + E)
- decide if an edge exists O(1)
- insert an edge O(1)
- delete an edge O(1)
- delete a vertex
- add a vertex

How much space is used?

V + E



# Graph Vocabulary -- Connected Graphs

**Connected graph** – a graph where every vertex is connected to every other vertex via some path. It is not required for every vertex to have an edge to every other vertex

There exists some way to get from each vertex to every other vertex

**Connected Component** – a *subgraph* in which any two vertices are connected via some path, but is connected to no additional vertices in the *supergraph*

- There exists some way to get from each vertex within the connected component to every other vertex in the connected component
- A vertex with no edges is itself a connected component