Introduction to Graphs

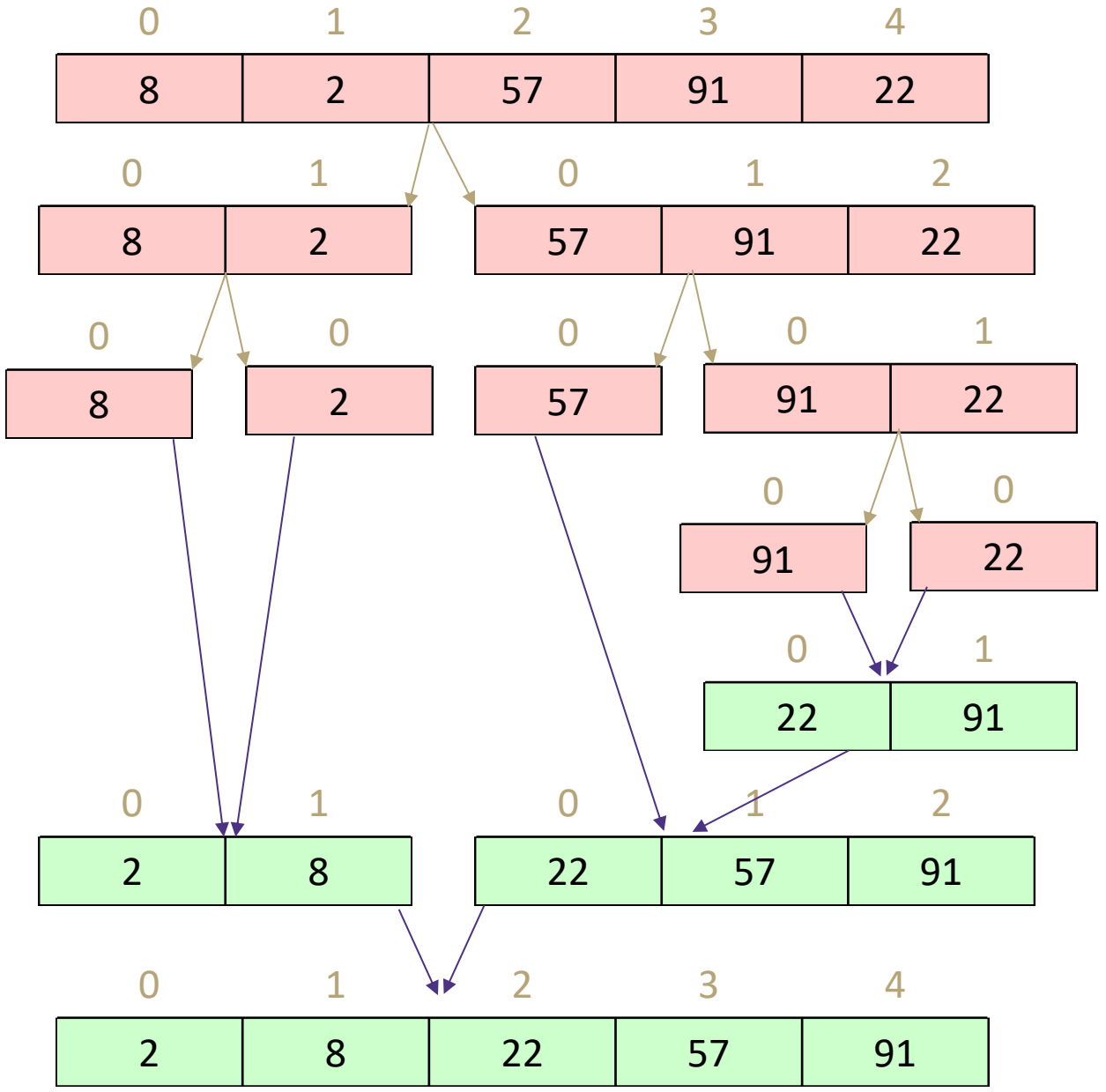
Data Structures and Algorithms

Warm Up

The Merge Sort algorithm has the same runtime for its worst, best and average case. Create a recurrence that represents this runtime:

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

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



Review: Unfolding Technique

$$T(n) = \begin{cases} 1 \; when \; n = 0 \\ 2T(n-1) + 1 \; otherwise \end{cases}$$

$$T(5) = 2T(5-1) + 1$$

$$= 2(2T(4-1) + 1) + 1$$

$$= 2(2(2T(3-1) + 1) + 1) + 1$$

$$= 2(2(2(2T(2-1) + 1) + 1) + 1) + 1$$

$$= 2(2(2(2(2T(1-1) + 1) + 1) + 1) + 1) + 1$$

$$= 2(2(2(2(2(1) + 1) + 1) + 1) + 1) + 1$$

$$= 2^5 + 5$$

$$= 2^n + n$$

The pattern by which we move towards the base case is n – 1
We would call this a "**linear recurrence**"

Unfolding Technique

$$T(n) = \begin{cases} 1 \; when \; n \leq 1 \\ 2T\left(\frac{n}{2}\right) + n \; otherwise \end{cases}$$

$$T(16) = 2T\left(\frac{16}{2}\right) + 16$$

$$= 2(2T\left(\frac{8}{2}\right) + 8) + 16$$

$$= 2(2(2T\left(\frac{4}{2}\right) + 4) + 8) + 16$$

$$= 2(2(2(2T\left(\frac{2}{2}\right) + 2) + 4) + 8) + 16$$

$$= 2(2(2(2(1) + 2) + 4) + 8) + 16$$

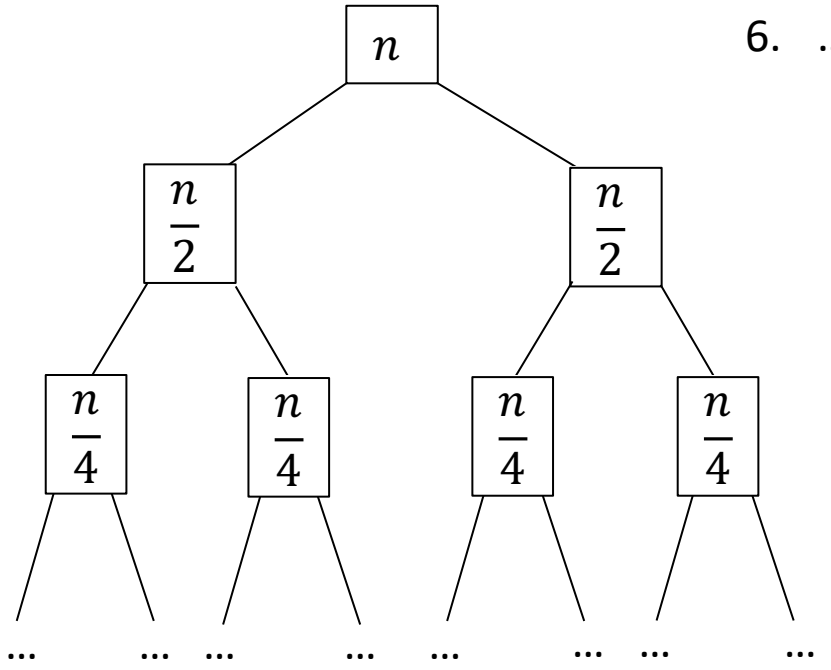
$$= 16 + 16 + 16 + 16 + 16 \qquad = 2^4 + \sum_{1}^{4} 16 \qquad = 2^{\sqrt{n}} + \sum_{1}^{\sqrt{n}} n$$

The pattern by which we move towards the base case is **not** linear
Multiple recursive calls cause **branching**
Is there an easier way to find the closed form?

Tree Method

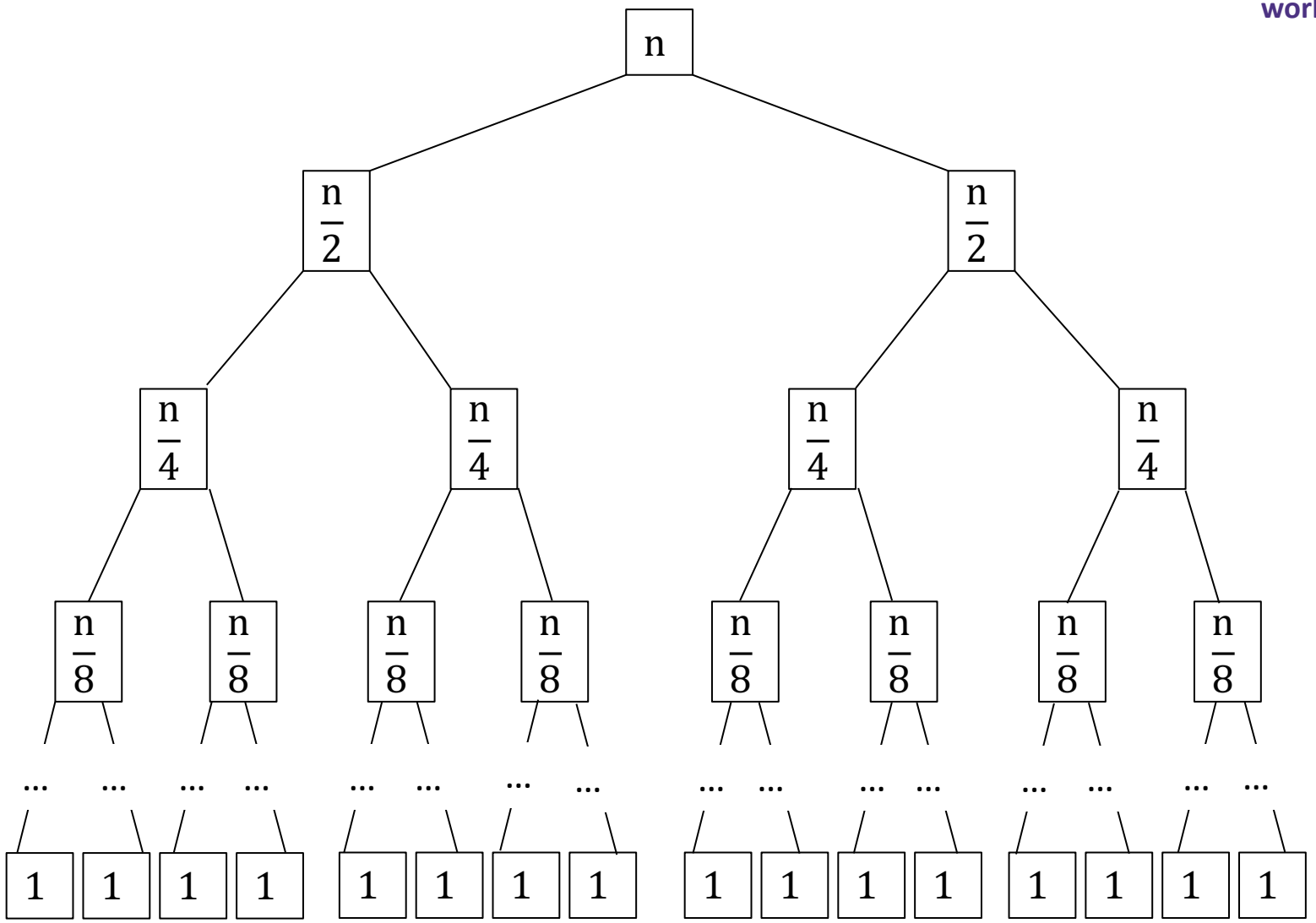
$$T(n) = \begin{cases} 1 \; when \; n \leq 1 \\ T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n \; otherwise \end{cases}$$

1. What work to do?
2. Replace with definition
3. Break apart non recursive and recursive pieces
4. Replace with definition
5. Break apart non recursive and recursive pieces
6. …



Tree Method

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



How many pieces of work at each level?	How much work at each level?
1	n
2	n
4	n
8	n
n	n

How much work per level?

n

How many levels?

log(n)

Runtime

$$T(n) = \sum_{i=0}^{\log_2 n} n$$

Tree Method Formulas

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

How much work is done by recursive levels (branch nodes)?

1. How many nodes are on each branch level i? branchNum(i)
   - i = 0 is overall root level
   - How many recursive calls are in each recursive branch to the power of which branch

   branchNum(i) = 2^i
2. At each level i, how much work does a single node do? branchWork(i)

   branchWork(i) = $(n/ 2^i)$
3. How many recursive levels are there? branchCount
   - Based on the pattern of how we get down to base case

   branchCount = $\log_2 n - 1$

$$Recursive\ work = \sum_{i=0}^{branchCount} branchNum(i)branchWork(i)$$

$$T(n > 1) = \sum_{i=0}^{\log_2 n - 1} 2^i \left(\frac{n}{2^i}\right)$$

How much work is done by the base case level (leaf nodes)?

1. How much work does a single leaf node do? leafWork

   leafWork = 1
2. How many leaf nodes are there? leafCount

   leafCount = $2^{\log 2n} = n$
   - How many branch nodes are in the second to last level x recursive calls per node

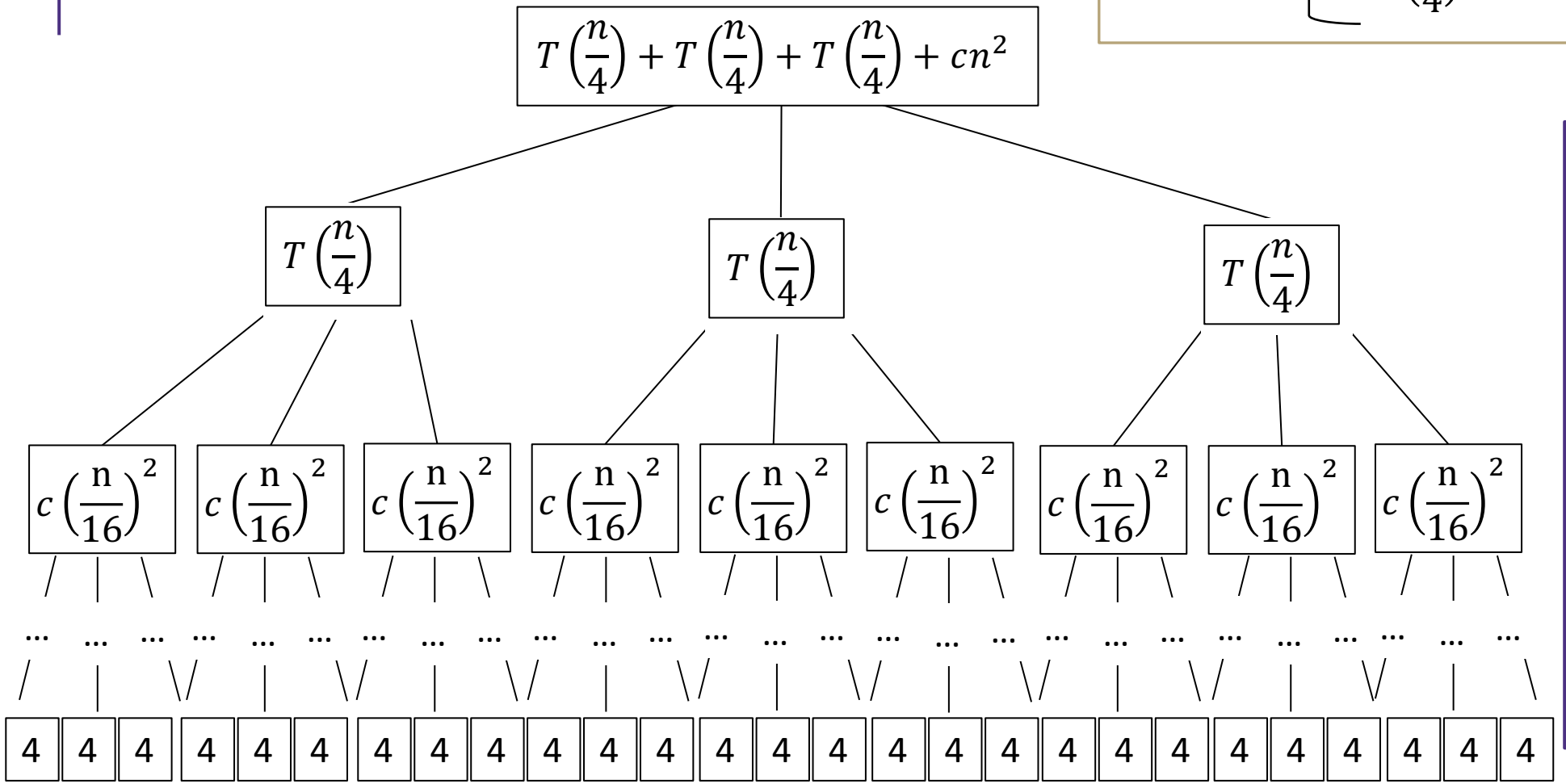
$$NonRecursive\ work = leafWork \times leafCount = leafWork \times branchNum^{numLevels}$$

$$T(n \leq 1) = n$$

$$T(n) = \sum_{i=0}^{\log_2 n - 1} 2^i \left(\frac{n}{2^i}\right) + n = n \log_2 n + n$$

Tree Method Practice

$$T(n) = \begin{cases} 4 \text{ when } n \leq 1 \\ 3T\left(\frac{n}{4}\right) + cn^2 \text{ otherwise} \end{cases}$$



Answer the following questions:

1. How many nodes on each branch level?
2. How much work for each branch node?
3. How much work per branch level?
4. How many branch levels?
5. How much work for each leaf node?
6. How many leaf nodes?

EXAMPLE PROVIDED BY CS 161 – JESSICA SU
HTTPS://WEB.STANFORD.EDU/CLASS/ARCHIVE/CS/CS161/CS161.1168/LECTURE3.PDF

Tree Method Practice

$$T(n) = \begin{cases} 4 \text{ when } n \leq 1 \\ 3T\left(\frac{n}{4}\right) + cn^2 \text{ otherwise} \end{cases}$$

1. How many nodes on each branch level? 3^i
2. How much work for each branch node? $c\left(\frac{n}{4^i}\right)^2$
3. How much work per branch level? $3^i c\left(\frac{n}{4^i}\right)^2 = \left(\frac{3}{16}\right)^i cn^2$
4. How many branch levels? $\log_4 n - 1$
5. How much work for each leaf node? 4
6. How many leaf nodes? $3^{\log_4 n}$ $n^{\log_4 3}$

power of a log

$x^{\log_b y} = y^{\log_b x}$

Level (i)	Number of Nodes	Work per Node	Work per Level
0	1	cn^2	cn^2
1	3	$c\left(\frac{n}{4}\right)^2$	$\frac{3}{16}cn^2$
2	9	$c\left(\frac{n}{16}\right)^2$	$\frac{9}{256}cn^2$
base	$3^{\log_4 n}$	4	$12^{\log_4 n}$

Combining it all together…

$$T(n) = \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + 4n^{\log_4 3}$$

Tree Method Practice

$$T(n) = \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + 4n^{\log_4 3}$$

factoring out a constant

$$\sum_{i=a}^{b} cf(i) = c\sum_{i=a}^{b} f(i)$$

$$T(n) = cn^2 \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i + 4n^{\log_4 3}$$

finite geometric series

$$\sum_{i=0}^{n-1} x^i = \frac{x^n - 1}{x - 1}$$

Closed form:

$$T(n) = cn^2 \left(\frac{\frac{3}{16}^{\log_4 n} - 1}{\frac{3}{16} - 1}\right) + 4n^{\log_4 3}$$

If we're trying to prove upper bound…

$$T(n) = cn^2 \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i + 4n^{\log_4 3}$$

infinite geometric series

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1 - x}$$

when -1 < x < 1

$$T(n) = cn^2 \left(\frac{1}{1 - \frac{3}{16}}\right) + 4n^{\log_4 3}$$

$$T(n) \in O(n^2)$$

Is there an easier way?

What if you don't want an exact closed form?

Sorry, no

If we want to find a big Θ

Yes!

Master Theorem

Given a recurrence of the following form:

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

Then thanks to magical math brilliance we can assume the following:

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 $T(n) \in \Theta\left(n^{\log_b a}\right)$

Apply Master Theorem

Given a recurrence of the form:

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

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 $T(n) \in \Theta\left(n^{\log_b a}\right)$

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

a = 2
b = 2
c = 1
d = 1

$$\log_b a = c \Rightarrow \log_2 2 = 1$$

$$T(n) \in \Theta(n^c \log_2 n) \Rightarrow \Theta(n^1 \log_2 n)$$

Reflecting on Master Theorem

Given a recurrence of the form:

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

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 $T(n) \in \Theta\left(n^{\log_b a}\right)$

$height \approx \log_b a$

$branchWork \approx n^c \log_b a$

$leafWork \approx d\left(n^{\log_b a}\right)$

The $\log_b a < c$ case

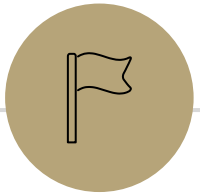
- Recursive case conquers work more quickly than it divides work
- Most work happens near "top" of tree
- Non recursive work in recursive case dominates growth, n^c term

The $\log_b a = c$ case

- Work is equally distributed across call stack (throughout the "tree")
- Overall work is approximately work at top level x height

The $\log_b a > c$ case

- Recursive case divides work faster than it conquers work
- Most work happens near "bottom" of tree
- Leaf work dominates branch work



Introduction to Graphs

Inter-data Relationships

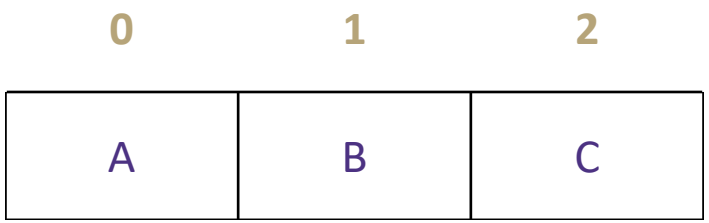
Arrays

Categorically associated

Sometimes ordered

Typically independent

Elements only store pure data, no connection info



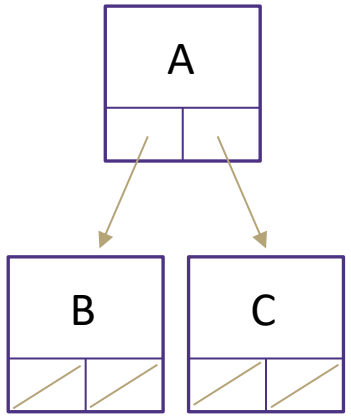
Trees

Directional Relationships

Ordered for easy access

Limited connections

Elements store data and connection info



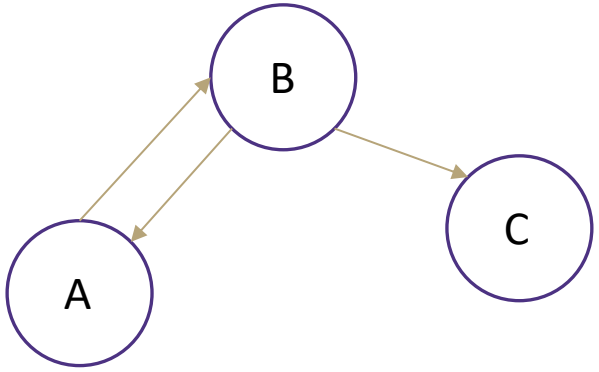
Graphs

Multiple relationship connections

Relationships dictate structure

Connection freedom!

Both elements and connections can store data



Graph: Formal Definition

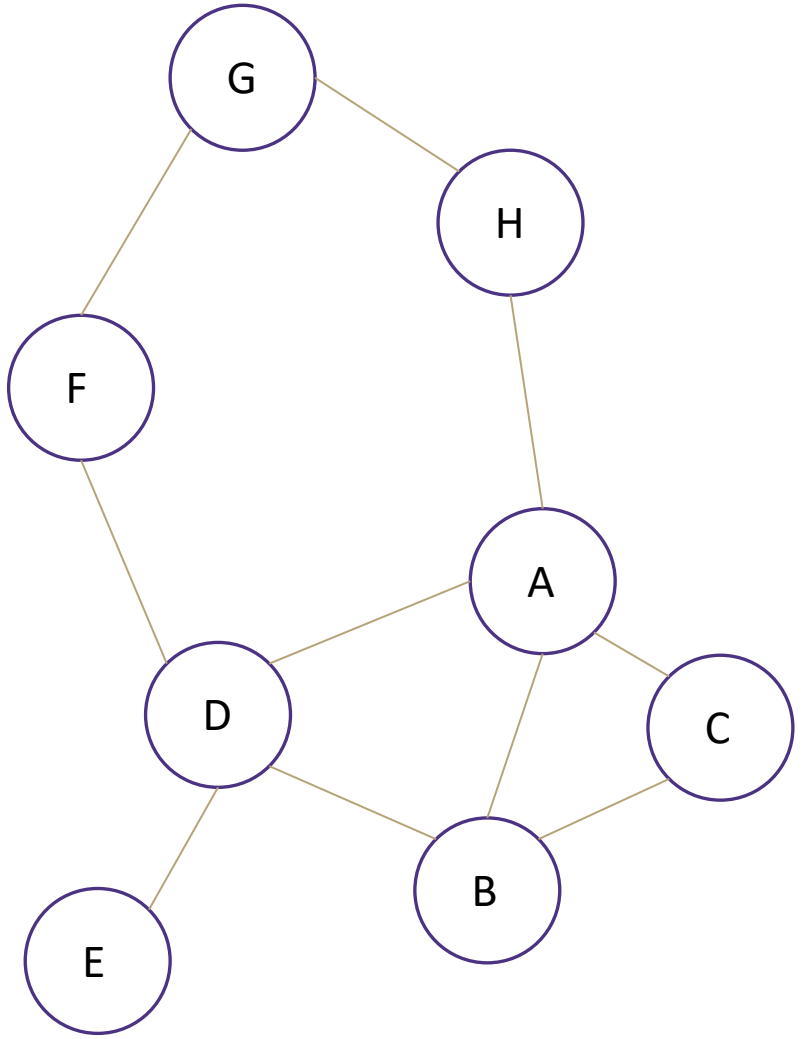
A **graph** is defined by a pair of sets G = (V, E) where…

- V is a set of **vertices**
  - A vertex or "node" is a data entity

V = { A, B, C, D, E, F, G, H }

- E is a set of **edges**
  - An edge is a connection between two vertices

E = { (A, B), (A, C), (A, D), (A, H),
(C, B), (B, D), (D, E), (D, F),
(F, G), (G, H)}



Applications

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

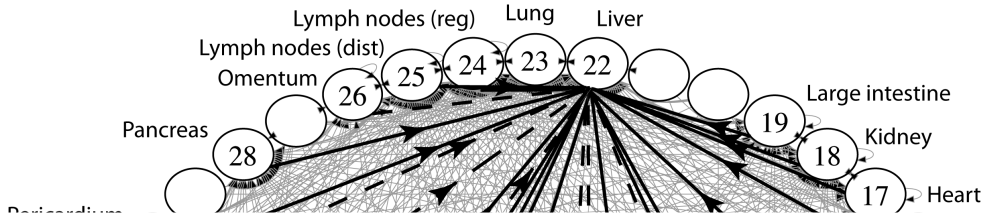
Relationships
- Social media graphs
  - Vertices are accounts, edges are follower relationships
- Code bases
  - Vertices are classes, edges are usage

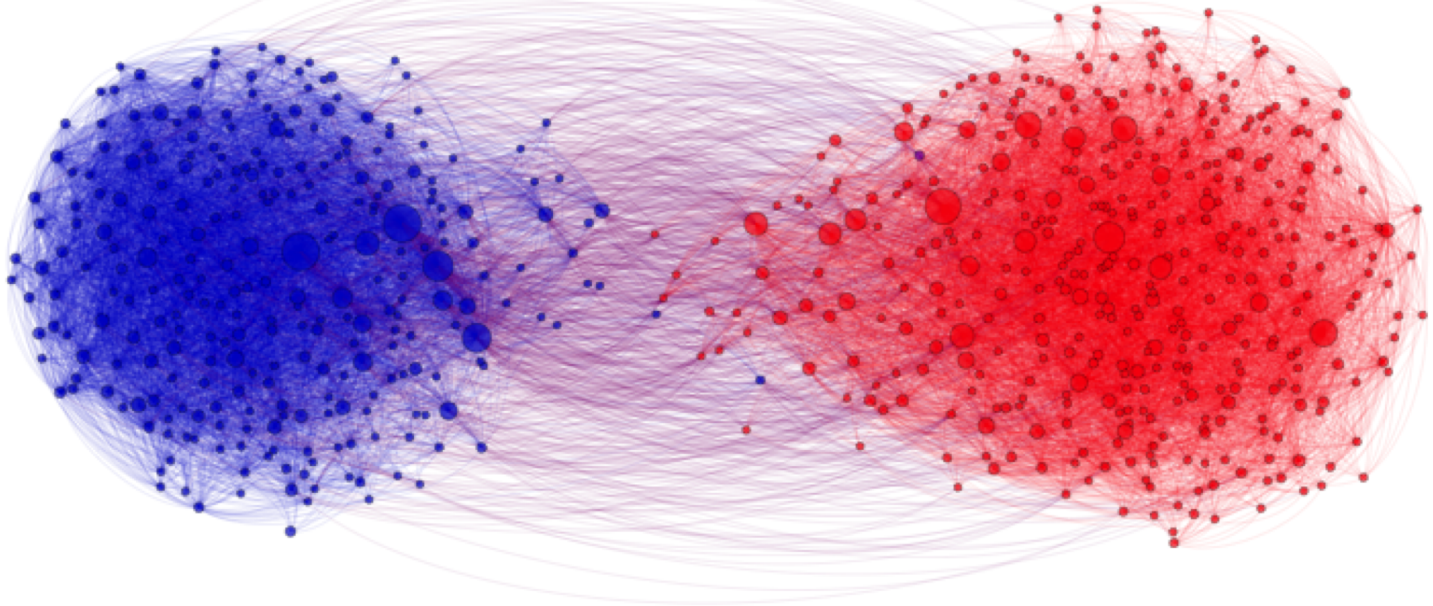
Influence
- Biology
  - Vertices are cancer cell destinations, edges are migration paths

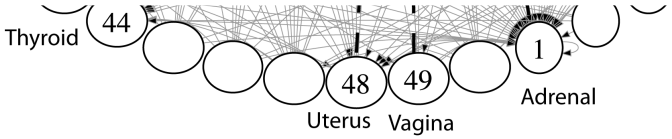
Related topics
- Web Page Ranking
  - Vertices are web pages, edges are hyperlinks
- Wikipedia
  - Vertices are articles, edges are links

SO MANY MORREEEE

www.allthingsgraphed.com







Graph Vocabulary

Graph Direction

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

  V = { A, B, C }

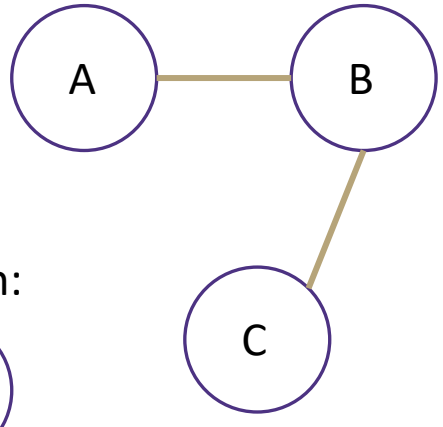
  E = { (A, B), (B, C) } *inferred (B, A) and (C,B)*

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

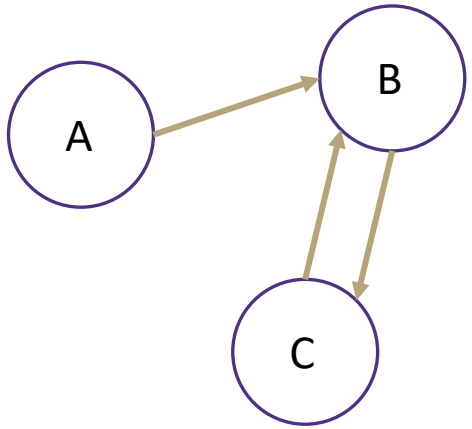
  V = { A, B, C }

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

Undirected Graph:



Undirected Graph:



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

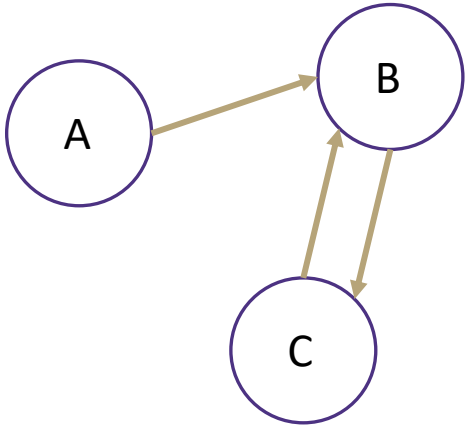
- **Out-degree** – the number of directed edges that start at a vertex

  A : 1, B : 1, C : 1

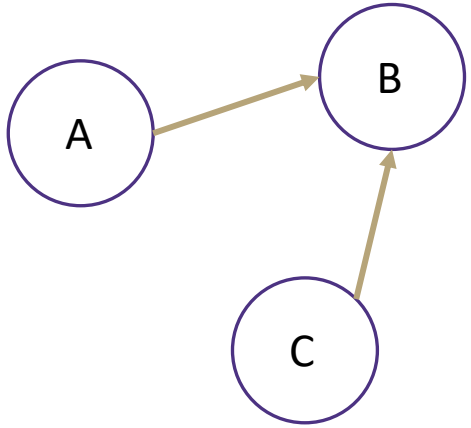
Food for thought

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

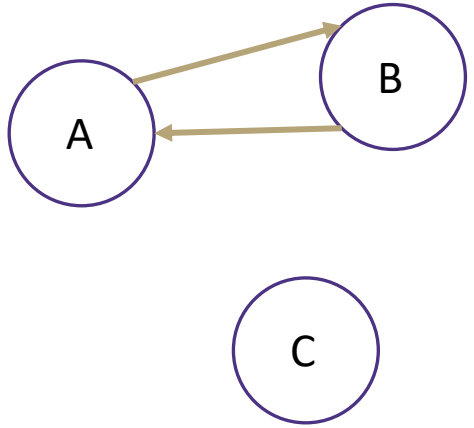
Yes



A has an "in degree" of 0

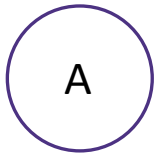


B has an "out degree" of 0



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

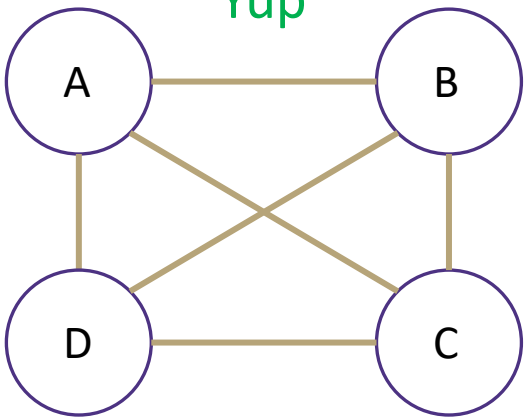
Is this a valid graph?

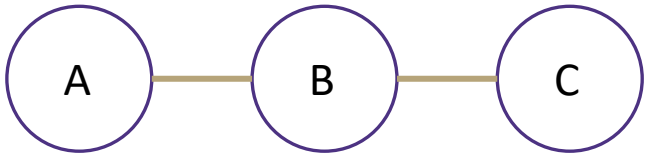


Yes!

Are these valid?

Yup

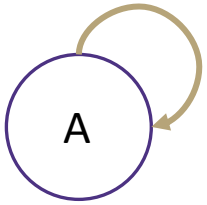




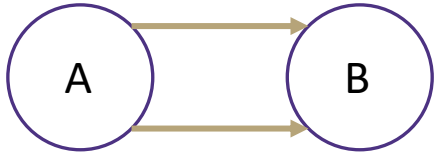
Sure

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