CSE 373: Master method, finishing sorts, intro to graphs

Michael Lee

Monday, Feb 12, 2018

The tree method: precise analysis

Problem: Need a rigorous way of getting a closed form

The tree method: precise analysis

Problem: Need a rigorous way of getting a closed form

We want to answer a few core questions:

The tree method: precise analysis

Problem: Need a rigorous way of getting a closed form

We want to answer a few core questions:

How much work does each recursive level do?

The tree method: precise analysis

Problem: Need a rigorous way of getting a closed form

We want to answer a few core questions:

How much work does each recursive level do?

1. How many nodes are there on level i? ($i = 0$ is "root" level)

The tree method: precise analysis

Problem: Need a rigorous way of getting a closed form

We want to answer a few core questions:

How much work does each recursive level do?

1. How many nodes are there on level *i*? ($i = 0$ is "root" level)
2. At some level *i*, how much work does a *single* node do? (Ignoring subtrees)

The tree method: precise analysis

Problem: Need a rigorous way of getting a closed form

We want to answer a few core questions:

How much work does each recursive level do?

1. How many nodes are there on level i? ($i = 0$ is "root" level)
2. At some level i, how much work does a *single* node do? (Ignoring subtrees)
3. How many recursive levels are there?

The tree method: precise analysis

Problem: Need a rigorous way of getting a closed form

We want to answer a few core questions:

How much work does each recursive level do?

1. How many nodes are there on level *i*? ($i = 0$ is "root" level)
2. At some level *i*, how much work does a *single* node do? (Ignoring subtrees)
3. How many recursive levels are there?

How much work does the leaf level (base cases) do?

The tree method: precise analysis

Problem: Need a rigorous way of getting a closed form

We want to answer a few core questions:

How much work does each recursive level do?

1. How many nodes are there on level *i*? ($i = 0$ is "root" level)
2. At some level *i*, how much work does a *single* node do? (Ignoring subtrees)
3. How many recursive levels are there?

How much work does the leaf level (base cases) do?

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

The tree method: precise analysis

Problem: Need a rigorous way of getting a closed form

We want to answer a few core questions:

How much work does each recursive level do?

1. How many nodes are there on level i? ($i = 0$ is "root" level)
2. At some level i, how much work does a *single* node do? (Ignoring subtrees)
3. How many recursive levels are there?

How much work does the leaf level (base cases) do?

1. How much work does a single leaf node do?
2. How many leaf nodes are there?

The tree method: precise analysis

Problem: Need a rigorous way of getting a closed form

We want to answer a few core questions:

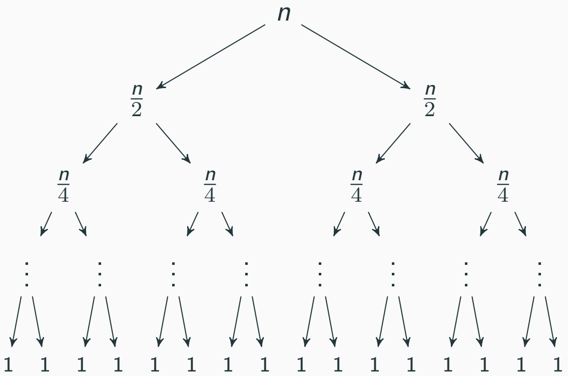
How much work does each recursive level do?

1. How many nodes are there on level *i*? ($i = 0$ is "root" level)
2. At some level *i*, how much work does a *single* node do? (Ignoring subtrees)
3. How many recursive levels are there?

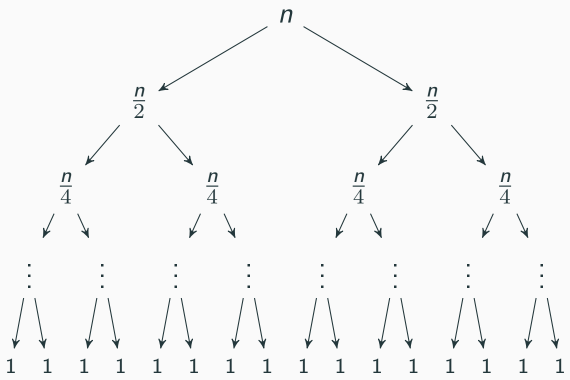
How much work does the leaf level (base cases) do?

1. How much work does a single leaf node do?
2. How many leaf nodes are there?

The tree method: precise analysis

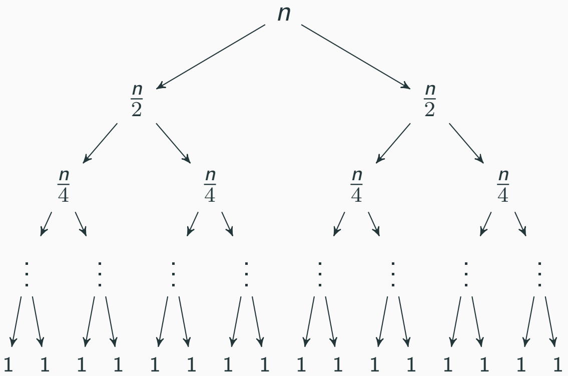


The tree method: precise analysis



1. numNodes(i) $= ?$
2. workPerNode(n, i) $= ?$
3. numLevels(n) $= ?$
4. workPerLeafNode(n) $= ?$
5. numLeafNodes(n) $= ?$

The tree method: precise analysis



1 node, n work per

2 nodes, $\frac{n}{2}$ work per

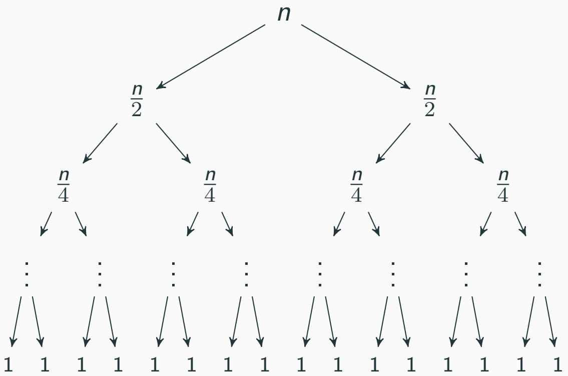
4 nodes, $\frac{n}{4}$ work per

2^i nodes, $\frac{n}{i}$ work per

2^h nodes, 1 work per

1. numNodes(i) $= ?$
2. workPerNode(n, i) $= ?$
3. numLevels(n) $= ?$
4. workPerLeafNode(n) $= ?$
5. numLeafNodes(n) $= ?$

The tree method: precise analysis



1 node, n work per

2 nodes, $\frac{n}{2}$ work per

4 nodes, $\frac{n}{4}$ work per

2^i nodes, $\frac{n}{i}$ work per

2^h nodes, 1 work per

1. numNodes(i) $= 2^i$
2. workPerNode(n, i) $= \frac{n}{2^i}$
3. numLevels(n) $= ?$
4. workPerLeafNode(n) $= 1$
5. numLeafNodes(n) $= ?$

The tree method: precise analysis

How many levels are there, exactly? Is it $\log_2(n)$?

The tree method: precise analysis

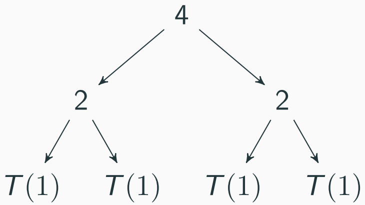
How many levels are there, exactly? Is it $\log_2(n)$?

Let's try an example. Suppose we have $T(4)$. What happens?

The tree method: precise analysis

How many levels are there, exactly? Is it $\log_2(n)$?

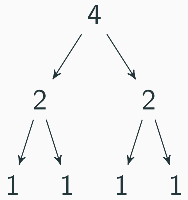
Let's try an example. Suppose we have $T(4)$. What happens?



The tree method: precise analysis

How many levels are there, exactly? Is it $\log_2(n)$?

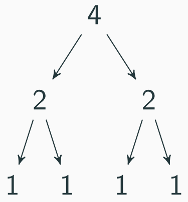
Let's try an example. Suppose we have $T(4)$. What happens?



The tree method: precise analysis

How many levels are there, exactly? Is it $\log_2(n)$?

Let's try an example. Suppose we have $T(4)$. What happens?



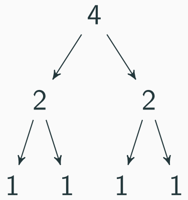
Height is $\log_2(4) = 2$.

For this recursive function, num recursive levels is same as height.

The tree method: precise analysis

How many levels are there, exactly? Is it $\log_2(n)$?

Let's try an example. Suppose we have $T(4)$. What happens?



Height is $\log_2(4) = 2$.

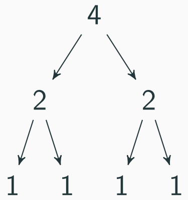
For this recursive function, num recursive levels is same as height.

Important: total levels, counting base case, is height $+ 1$.

The tree method: precise analysis

How many levels are there, exactly? Is it $\log_2(n)$?

Let's try an example. Suppose we have $T(4)$. What happens?



Height is $\log_2(4) = 2$.

For this recursive function, num recursive levels is same as height.

Important: total levels, counting base case, is height $+ 1$.

Important: for other recursive functions, where base case doesn't happen at $n \leq 1$, num recursive levels might be different then

The tree method: precise analysis

We discovered:

1. $\text{numNodes}(i) = 2^i$
2. $\text{workPerNode}(n, i) = \frac{n}{2^i}$
3. $\text{numLevels}(n) = \log_2(n)$
4. $\text{workPerLeafNode}(n) = 1$
5. $\text{numLeafNodes}(n) = 2^{\text{numLevels}(n)} = 2^{\log_2(n)} = n$

The tree method: precise analysis

We discovered:

1. $\text{numNodes}(i) = 2^i$
2. $\text{workPerNode}(n, i) = \frac{n}{2^i}$
3. $\text{numLevels}(n) = \log_2(n)$
4. $\text{workPerLeafNode}(n) = 1$
5. $\text{numLeafNodes}(n) = 2^{\text{numLevels}(n)} = 2^{\log_2(n)} = n$

Our formulas:

$$\text{recursiveWork} = \sum_{i=0}^{\text{numLevels(n)}} \text{numNodes}(i) \cdot \text{workPerNode}(n, i)$$

The tree method: precise analysis

We discovered:

1. $\text{numNodes}(i) = 2^i$
2. $\text{workPerNode}(n, i) = \frac{n}{2^i}$
3. $\text{numLevels}(n) = \log_2(n)$
4. $\text{workPerLeafNode}(n) = 1$
5. $\text{numLeafNodes}(n) = 2^{\text{numLevels}(n)} = 2^{\log_2(n)} = n$

Our formulas:

$$\text{recursiveWork} = \sum_{i=0}^{\text{numLevels(n)}} \text{numNodes}(i) \cdot \text{workPerNode}(n, i)$$

$$\text{baseCaseWork} = \text{numLeafNodes}(n) \cdot \text{workPerLeafNode}(n)$$

The tree method: precise analysis

We discovered:

1. $\text{numNodes}(i) = 2^i$
2. $\text{workPerNode}(n, i) = \frac{n}{2^i}$
3. $\text{numLevels}(n) = \log_2(n)$
4. $\text{workPerLeafNode}(n) = 1$
5. $\text{numLeafNodes}(n) = 2^{\text{numLevels}(n)} = 2^{\log_2(n)} = n$

Our formulas:

$$\text{recursiveWork} = \sum_{i=0}^{\text{numLevels(n)}} \text{numNodes}(i) \cdot \text{workPerNode}(n, i)$$

$$\text{baseCaseWork} = \text{numLeafNodes}(n) \cdot \text{workPerLeafNode}(n)$$

$$\text{totalWork} = \text{recursiveWork} + \text{baseCaseWork}$$

The tree method: precise analysis

Solve for recursive case:

$$\text{recursiveWork} = \sum_{i=0}^{\log_2(n)} 2^i \cdot \frac{n}{2^i}$$

The tree method: precise analysis

Solve for recursive case:

$$\text{recursiveWork} = \sum_{i=0}^{\log_2(n)} 2^i \cdot \frac{n}{2^i}$$

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

The tree method: precise analysis

Solve for recursive case:

$$\begin{aligned}\text{recursiveWork} &= \sum_{i=0}^{\log_2(n)} 2^i \cdot \frac{n}{2^i} \\ &= \sum_{i=0}^{\log_2(n)} n \\ &= n \log_2(n)\end{aligned}$$

Solve for base case:

$$\begin{aligned}\text{baseCaseWork} &= \text{numLeafNodes}(n) \cdot \text{workDonePerLeafNode}(n) \\ &= n \cdot 1 = n\end{aligned}$$

The tree method: precise analysis

Solve for recursive case:

$$\begin{aligned}\text{recursiveWork} &= \sum_{i=0}^{\log_2(n)} 2^i \cdot \frac{n}{2^i} \\ &= \sum_{i=0}^{\log_2(n)} n \\ &= n \log_2(n)\end{aligned}$$

Solve for base case:

$$\begin{aligned}\text{baseCaseWork} &= \text{numLeafNodes}(n) \cdot \text{workDonePerLeafNode}(n) \\ &= n \cdot 1 = n\end{aligned}$$

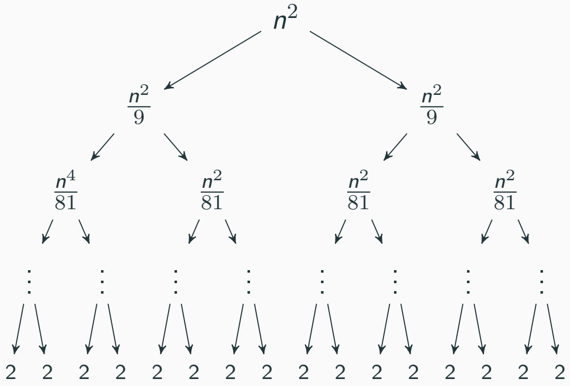
So exact closed form is $n \log_2(n) + n$.

The tree method: practice

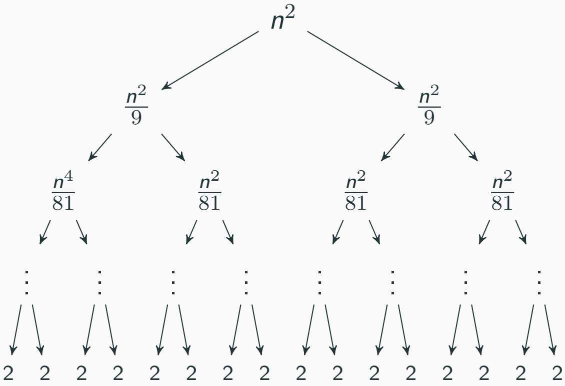
Practice: Let's go back to our old recurrence...

$$S(n) = \begin{cases} 2 & \text{if } n \leq 1 \\ 2S\left(n/3\right) + n^2 & \text{otherwise} \end{cases}$$

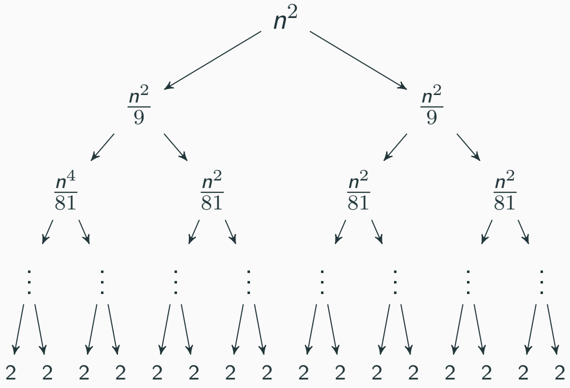
The tree method: practice



The tree method: practice



1. numNodes(i) $= ?$
2. workPerNode(n, i) $= ?$
3. numLevels(n) $= ?$
4. workPerLeafNode(n) $= ?$
5. numLeafNodes(n) $= ?$



1 node, n^2 work per

2 nodes, $\frac{n^2}{3^2}$ work per

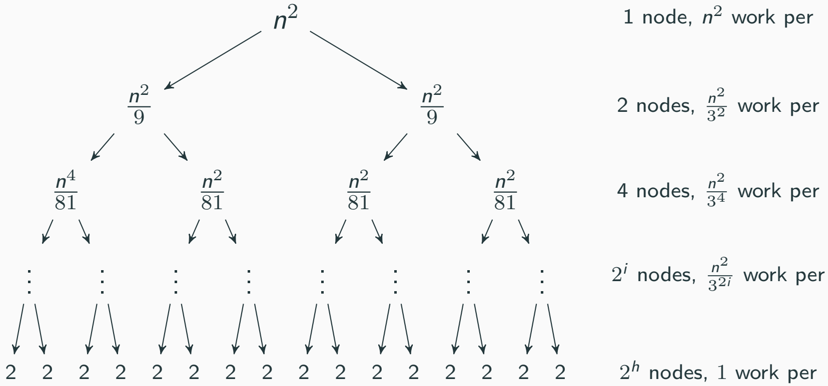
4 nodes, $\frac{n^2}{3^4}$ work per

2^i nodes, $\frac{n^2}{3^{2i}}$ work per

2^h nodes, 1 work per

1. numNodes(i) $= ?$
2. workPerNode(n, i) $= ?$
3. numLevels(n) $= ?$
4. workPerLeafNode(n) $= ?$
5. numLeafNodes(n) $= ?$

The tree method: practice



1. numNodes(i) $= 2^i$
2. workPerNode(n, i) $= \frac{n^2}{9^i}$
3. numLevels(n) $= \log_3(n)$
4. workPerLeafNode(n) $= 2$
5. numLeafNodes(n) $= 2^{\text{numLevels}(n)} = 2^{\log_3(n)} = n^{\log_3(2)}$

The tree method: practice

1. numNodes(i) $= 2^i$
2. workPerNode(n, i) $= \frac{n^2}{9^i}$
3. numLevels(n) $= \log_3(n)$
4. workPerLeafNode(n) $= 2$
5. numLeafNodes(n) $= 2^{\text{numLevels}(n)} = 2^{\log_3(n)} = n^{\log_3(2)}$

Combine into a single expression representing the total runtime.

The tree method: practice

1. $\text{numNodes}(i) = 2^i$
2. $\text{workPerNode}(n, i) = \frac{n^2}{9^i}$
3. $\text{numLevels}(n) = \log_3(n)$
4. $\text{workPerLeafNode}(n) = 2$
5. $\text{numLeafNodes}(n) = 2^{\text{numLevels}(n)} = 2^{\log_3(n)} = n^{\log_3(2)}$

Combine into a single expression representing the total runtime.

$$\text{totalWork} = \left(\sum_{i=0}^{\log_3(n)} 2^i \cdot \frac{n^2}{9^i} \right) + 2n^{\log_3(2)}$$

The tree method: practice

1. $\text{numNodes}(i) = 2^i$
2. $\text{workPerNode}(n, i) = \frac{n^2}{9^i}$
3. $\text{numLevels}(n) = \log_3(n)$
4. $\text{workPerLeafNode}(n) = 2$
5. $\text{numLeafNodes}(n) = 2^{\text{numLevels}(n)} = 2^{\log_3(n)} = n^{\log_3(2)}$

Combine into a single expression representing the total runtime.

$$\begin{aligned}\text{totalWork} &= \left(\sum_{i=0}^{\log_3(n)} 2^i \cdot \frac{n^2}{9^i}\right) + 2n^{\log_3(2)} \\ &= n^2 \sum_{i=0}^{\log_3(n)} \frac{2^i}{9^i} + 2n^{\log_3(2)}\end{aligned}$$

The tree method: practice

1. $\text{numNodes}(i) = 2^i$
2. $\text{workPerNode}(n, i) = \frac{n^2}{9^i}$
3. $\text{numLevels}(n) = \log_3(n)$
4. $\text{workPerLeafNode}(n) = 2$
5. $\text{numLeafNodes}(n) = 2^{\text{numLevels}(n)} = 2^{\log_3(n)} = n^{\log_3(2)}$

Combine into a single expression representing the total runtime.

$$
\begin{aligned}
\text{totalWork} &= \left(\sum_{i=0}^{\log_3(n)} 2^i \cdot \frac{n^2}{9^i} \right) + 2n^{\log_3(2)} \\
&= n^2 \sum_{i=0}^{\log_3(n)} \frac{2^i}{9^i} + 2n^{\log_3(2)} \\
&= n^2 \sum_{i=0}^{\log_3(n)} \left(\frac{2}{9} \right)^i + 2n^{\log_3(2)}
\end{aligned}
$$

The finite geometric series

We have: $n^2 \sum_{i=0}^{\log_3(n)} \left(\frac{2}{9}\right)^i + 2n^{\log_3(2)}$

The finite geometric series

We have: $n^2 \sum_{i=0}^{\log_3(n)} \left(\frac{2}{9}\right)^i + 2n^{\log_3(2)}$

The finite geometric series identity: $\sum_{i=0}^{n-1} r^i = \frac{1 - r^n}{1 - r}$

The finite geometric series

We have: $n^2 \sum_{i=0}^{\log_3(n)} \left(\frac{2}{9}\right)^i + 2n^{\log_3(2)}$

The finite geometric series identity: $\sum_{i=0}^{n-1} r^i = \frac{1-r^n}{1-r}$

Plug and chug:

$$\begin{aligned}
\text{totalWork} &= n^2 \sum_{i=0}^{\log_3(n)} \left(\frac{2}{9}\right)^i + 2n^{\log_3(2)} \\
&= n^2 \sum_{i=0}^{\log_3(n)+1-1} \left(\frac{2}{9}\right)^i + 2n^{\log_3(2)} \\
&= n^2 \frac{1-\left(\frac{2}{9}\right)^{\log_3(n)+1}}{1-\frac{2}{9}} + 2n^{\log_3(2)}
\end{aligned}$$

Applying the finite geometric series

With a bunch of effort...

$$
\begin{aligned}
\text{totalWork} &= n^2 \frac{1 - \left(\frac{2}{9}\right)^{\log_3(n)+1}}{1 - \frac{2}{9}} + 2n^{\log_3(2)} \\
&= \frac{9}{7} n^2 \left(1 - \frac{2}{9}\left(\frac{2}{9}\right)^{\log_3(n)}\right) + 2n^{\log_3(2)} \\
&= \frac{9}{7} n^2 - \frac{2}{7} n^2 \left(\frac{2}{9}\right)^{\log_3(n)} + 2n^{\log_3(2)} \\
&= \frac{9}{7} n^2 - \frac{2}{7} n^2 n^{\log_3(2/9)} + 2n^{\log_3(2)} \\
&= \frac{9}{7} n^2 - \frac{2}{7} n^2 n^{\log_3(2)-2} + 2n^{\log_3(2)} \\
&= \frac{9}{7} n^2 - \frac{2}{7} n^{\log_3(2)} + 2n^{\log_3(2)} \\
&= \frac{9}{7} n^2 + \frac{12}{7} n^{\log_3(2)}
\end{aligned}
$$

The master theorem

Is there an easier way?

The master theorem

Is there an easier way?

If we want to find an exact closed form, no. Must use either the unfolding technique or the tree technique.

The master theorem

Is there an easier way?

If we want to find an exact closed form, no. Must use either the unfolding technique or the tree technique.

If we want to find a big-Θ bound, yes.

The master theorem

Suppose we have a recurrence of the following form:

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

The master theorem

Suppose we have a recurrence of the following form:

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

Then...

- If $\log_b(a) < c$, then $T(n) \in \Theta\left(n^c\right)$
- If $\log_b(a) = c$, then $T(n) \in \Theta\left(n^c \log(n)\right)$
- If $\log_b(a) > c$, then $T(n) \in \Theta\left(n^{\log_b(a)}\right)$

The master theorem

Given:

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

Then...

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

If $\log_b(a) = c$, then $T(n) \in \Theta\left(n^c \log(n)\right)$

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

The master theorem

Given:

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

Then...

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

If $\log_b(a) = c$, then $T(n) \in \Theta\left(n^c \log(n)\right)$

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

Sanity check: try checking merge sort.

The master theorem

Given:

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

Then...

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

If $\log_b(a) = c$, then $T(n) \in \Theta\left(n^c \log(n)\right)$

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

Sanity check: try checking merge sort.

We have $a = 2$, $b = 2$, and $c = 1$. We know $\log_b(a) = \log_2(2) = 1 = c$, therefore merge sort is $\Theta\left(n \log(n)\right)$.

The master theorem

Given:

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

Then...

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

If $\log_b(a) = c$, then $T(n) \in \Theta\left(n^c \log(n)\right)$

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

Sanity check: try checking merge sort.

We have $a = 2$, $b = 2$, and $c = 1$. We know $\log_b(a) = \log_2(2) = 1 = c$, therefore merge sort is $\Theta\left(n \log(n)\right)$.

Sanity check: try checking $S(n) = 2S(n/3) + n^2$.

The master theorem

Given:

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

Then...

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

If $\log_b(a) = c$, then $T(n) \in \Theta\left(n^c \log(n)\right)$

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

Sanity check: try checking merge sort.

We have $a = 2$, $b = 2$, and $c = 1$. We know $\log_b(a) = \log_2(2) = 1 = c$, therefore merge sort is $\Theta\left(n \log(n)\right)$.

Sanity check: try checking $S(n) = 2S(n/3) + n^2$.

We have $a = 2$, $b = 3$, and $c = 2$. We know $\log_3(2) \leq 1 < 2 = c$, therefore $S(n) \in \Theta\left(n^2\right)$.

The master theorem: intuition

Intuition, the $\log_b(a) < c$ case:

1. We do work more rapidly then we divide.
2. So, more of the work happens near the "top", which means that the n^c term dominates.

The master theorem: intuition

Intuition, the $\log_b(a) > c$ case:

1. We divide more rapidly then we do work.
2. So, most of the work happens near the "bottom", which means the work done in the leaves dominates.
3. Note: Work in leaves is about $d \cdot a^{\text{height}} = d \cdot a^{\log_b(n)} = d \cdot n^{\log_b(a)}$.

Intuition, the $\log_b(a) = c$ case:

1. Work is done roughly equally throughout tree.
2. Each level does about the same amount of work, so we approximate by just multiplying work done on first level by the height: $n^c \log_b(n)$.

Parting thoughts on sorting

A few final thoughts about sorting...

Hybrid sorts

Problem: Quick sort, in the best case, is pretty fast – often a constant factor faster then merge sort. But in the worst case, it's $\mathcal{O}\left(n^2\right)$!

Hybrid sorts

Problem: Quick sort, in the best case, is pretty fast – often a constant factor faster then merge sort. But in the worst case, it's $\mathcal{O}\left(n^2\right)$!

Idea: If things start looking bad, stop using quicksort! Switch to a different sorting algorithm.

Hybrid sorts

Problem: Quick sort, in the best case, is pretty fast – often a constant factor faster then merge sort. But in the worst case, it's $\mathcal{O}\left(n^2\right)$!

Idea: If things start looking bad, stop using quicksort! Switch to a different sorting algorithm.

Hybrid sort

A sorting algorithm which combines two or more other sorting algorithms.

Example: Introsort

Core idea: Combine quick sort with heap sort

(Why heap sort? It's also $\mathcal{O}\left(n \log(n)\right)$, and can be implemented in-place.)

Example: Introsort

Core idea: Combine quick sort with heap sort

(Why heap sort? It's also $\mathcal{O}(n \log(n))$, and can be implemented in-place.)

1. Run quicksort, but keep track of how many recursive calls we've made
2. If we pass some threshold (usually, $2\lfloor \lg(n) \rfloor$)

Example: Introsort

Core idea: Combine quick sort with heap sort

(Why heap sort? It's also $\mathcal{O}\left(n \log(n)\right)$, and can be implemented in-place.)

1. Run quicksort, but keep track of how many recursive calls we've made
2. If we pass some threshold (usually, $2\lfloor \lg(n) \rfloor$)
    2.1 Assume we've hit our worst case and switch to heapsort
    2.2 Else continue using quick sort

Example: Introsort

Core idea: Combine quick sort with heap sort

(Why heap sort? It's also $\mathcal{O}(n \log(n))$, and can be implemented in-place.)

1. Run quicksort, but keep track of how many recursive calls we've made
2. If we pass some threshold (usually, $2\lfloor \lg(n) \rfloor$)
    2.1 Assume we've hit our worst case and switch to heapsort
    2.2 Else continue using quick sort

Punchline: worst-case runtime is now $\mathcal{O}(n \log(n))$, not $\mathcal{O}\left(n^2\right)$.

Example: Introsort

Core idea: Combine quick sort with heap sort

(Why heap sort? It's also $\mathcal{O}(n\log(n))$, and can be implemented in-place.)

1. Run quicksort, but keep track of how many recursive calls we've made
2. If we pass some threshold (usually, $2\lfloor\lg(n)\rfloor$)
    2.1 Assume we've hit our worst case and switch to heapsort
    2.2 Else continue using quick sort

Punchline: worst-case runtime is now $\mathcal{O}(n\log(n))$, not $\mathcal{O}\left(n^2\right)$.

Used by various C++ implementations, used by Microsoft's .NET framework.

Adaptive sorts

Observation: Most real-world data is contains "mostly-sorted runs" – chunks of data that are already sorted.

Adaptive sorts

Observation: Most real-world data is contains "mostly-sorted runs" – chunks of data that are already sorted.

Idea: Modify our strategy based on what the input data actually looks like!

Adaptive sorts

Observation: Most real-world data is contains "mostly-sorted runs" – chunks of data that are already sorted.

Idea: Modify our strategy based on what the input data actually looks like!

Adaptive sort

A sorting algorithm that adapts to patterns and pre-existing order in the input.

Most adaptive sorts take advantage of how real-world data is often partially sorted.

Example: Timsort

Core idea: Combine merge sort with insertion sort

(Who's Tim? A Python core developer)

Example: Timsort

Core idea: Combine merge sort with insertion sort

(Who's Tim? A Python core developer)

1. Works by looking for "sorted runs".
2. Will use insertion sort to merge two small runs and merge sort to merge two large runs

Example: Timsort

Core idea: Combine merge sort with insertion sort

(Who's Tim? A Python core developer)

1. Works by looking for "sorted runs".
2. Will use insertion sort to merge two small runs and merge sort to merge two large runs
3. Implementation is very complex – lots of nuances, lots of clever tricks (e.g. detecting runs that are in reverse sorted order, sometimes skipping elements)

Example: Timsort

Core idea: Combine merge sort with insertion sort

(Who's Tim? A Python core developer)

1. Works by looking for "sorted runs".
2. Will use insertion sort to merge two small runs and merge sort to merge two large runs
3. Implementation is very complex – lots of nuances, lots of clever tricks (e.g. detecting runs that are in reverse sorted order, sometimes skipping elements)

Punchline: worst-case runtime is still $\mathcal{O}(n \log(n))$, but best case runtime is $\mathcal{O}(n)$.

Example: Timsort

Core idea: Combine merge sort with insertion sort

(Who's Tim? A Python core developer)

1. Works by looking for "sorted runs".
2. Will use insertion sort to merge two small runs and merge sort to merge two large runs
3. Implementation is very complex – lots of nuances, lots of clever tricks (e.g. detecting runs that are in reverse sorted order, sometimes skipping elements)

Punchline: worst-case runtime is still $\mathcal{O}\left(n \log(n)\right)$, but best case runtime is $\mathcal{O}\left(n\right)$.

Used by Python and Java.

Linear sorts (aka 'Niche' sorts)

Basic idea: Can we do better then $\mathcal{O}\left(n \log(n)\right)$ if we assume more things about the input list?

Counting sort

- **Assumption:** Input is a list of ints where every item is between 0 and k
- **Worst-case runtime:**

How would you implement this? Hint: start by creating a temp array of length k. Take inspiration from how hash tables work.

Counting sort

- **Assumption:** Input is a list of ints where every item is between 0 and k
- **Worst-case runtime:**

How would you implement this? Hint: start by creating a temp array of length k. Take inspiration from how hash tables work.

The algorithm:

1. Create a temp array named `arr` of length k.
2. Loop through the input array. For each number num, run `arr[num] += 1`
3. The temp array will now contain how many times we say each number in the input list.
4. Iterate through it to create the output list.

Counting sort

- **Assumption:** Input is a list of ints where every item is between 0 and k
- **Worst-case runtime:** $\mathcal{O}(n + k)$

How would you implement this? Hint: start by creating a temp array of length k. Take inspiration from how hash tables work.

The algorithm:

1. Create a temp array named `arr` of length k.
2. Loop through the input array. For each number num, run `arr[num] += 1`
3. The temp array will now contain how many times we say each number in the input list.
4. Iterate through it to create the output list.

Other linear sorts

Other interesting linear sorts:

- **Radix sort**
  - Assumes items in list are some kind of sequence-like thing such as strings or ints (which is a sequence of digits)
  - Assumes each "digit" is also sortable
  - Sorts all the digits in the 1s place, then the 2s place...

Other linear sorts

Other interesting linear sorts:

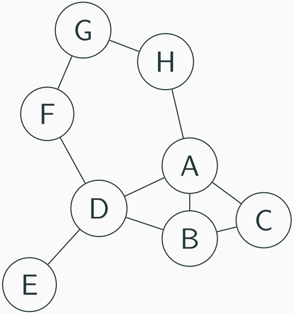
- **Radix sort**
  - Assumes items in list are some kind of sequence-like thing such as strings or ints (which is a sequence of digits)
  - Assumes each “digit” is also sortable
  - Sorts all the digits in the 1s place, then the 2s place...
- **Bucket sort**
  - A generalization of counting sort
  - Assumes items are randomly and uniformly distributed across a range of possibilities

Introduction to graphs

What is a graph?

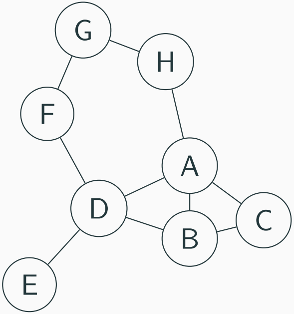
What is a graph?

This is a graph:

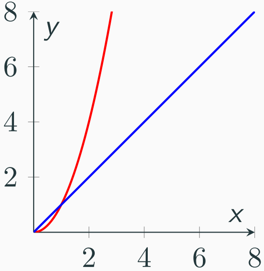


What is a graph?

This is a graph:



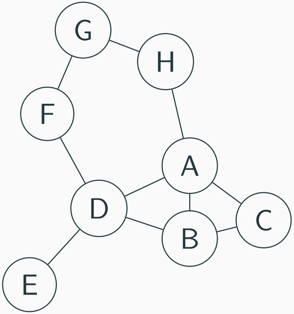
This is also a graph:



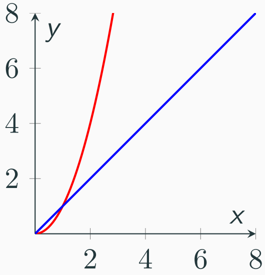
What is a graph?

What is a graph?

This is a graph:



This is also a graph:



In this class, by "graph", we mean the graph *data structure*.

Graph: formal definition

Graph

A **graph** is a pair $G = (V, E)$, where...

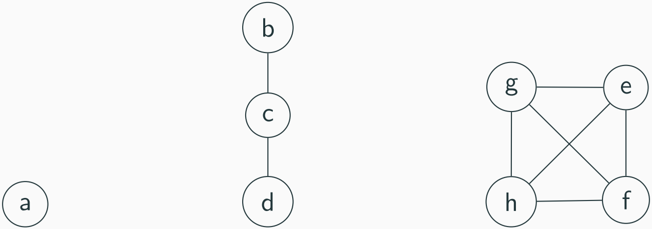
- V is a set of **vertices**
- E is a set of **edges** (pairs of vertices)

Notes:

- Vertices are the circle things, edges are the lines
- The words "node" and "vertex" are synonyms

Graph: formal definition examples

Examples:



$$V = \{a\} \qquad E = \{\}$$

$$V = \{b, c, d\} \qquad E = \{(b, c), (c, d)\}$$

$$V = \{e, f, g, h\}$$
$$E = \{(e, f), (f, g), (g, h), (h, e), (e, g), (f, h)\}$$

Applications of graphs

In a nutshell:

Graphs let us model the "relationship" between items.

If that seems like a very general definition, that's because graphs are a very general concept!

Applications of graphs

In a nutshell:

Graphs let us model the "relationship" between items.

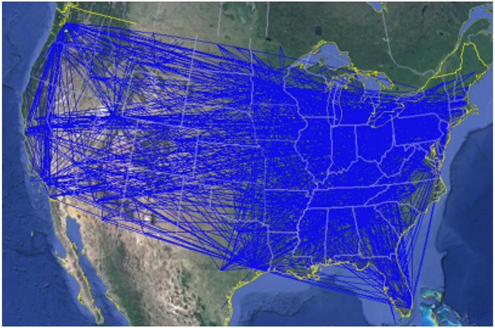
If that seems like a very general definition, that's because graphs are a very general concept!

Core insight:

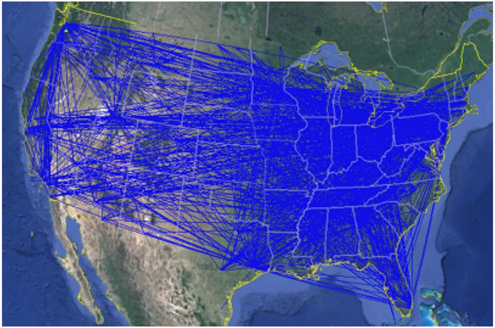
- Graphs are an abstract concept that appear in many different ways
- Many problems can be modeled as a graph problem

Some examples...

Application: Airline flight graph



Application: Airline flight graph



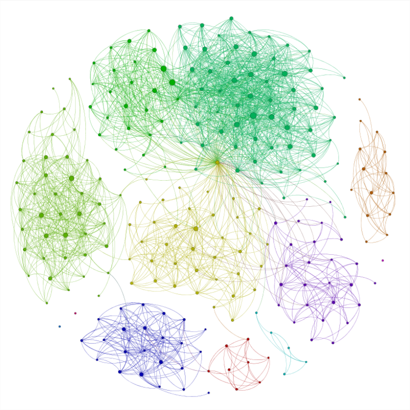
Questions: What is the cheapest/shortest/etc flight from A to B?

Is the route the airline offering me actually the cheapest route?

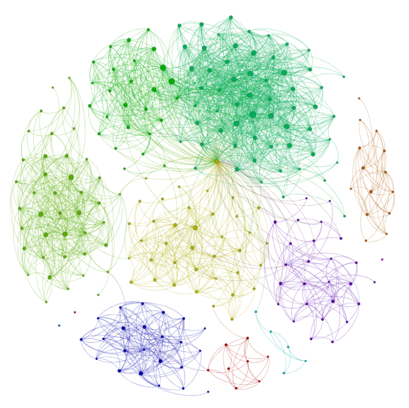
What happens if a city is snowed in – how can we reroute flights?

http://allthingsgraphed.com/public/images/airline-google-earth.png

Application: Social media graph



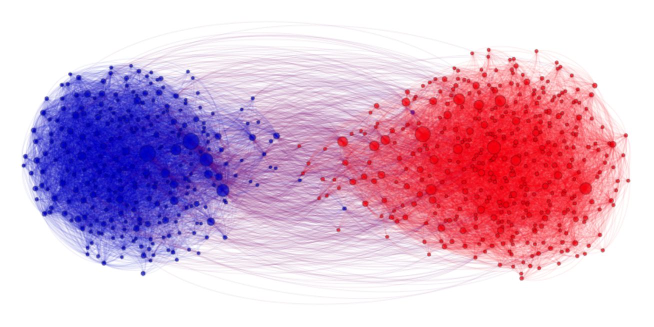
Application: Social media graph



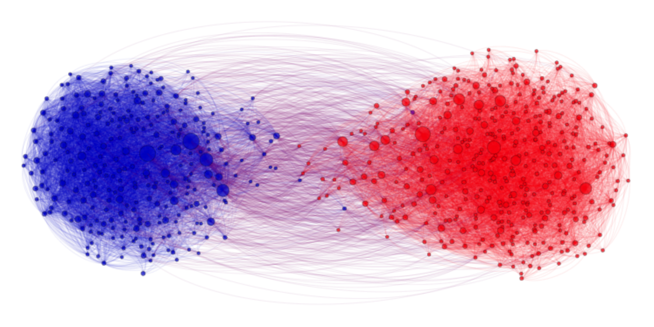
Questions: Why does this graph look clustered? Why are all my friends more popular then me? Who do my friends know? If I want to hire somebody to promote my product, who do I pick?

http://allthingsgraphed.com/public/images/facebook/facebook-friend-graph.png

Application: Social media polarization



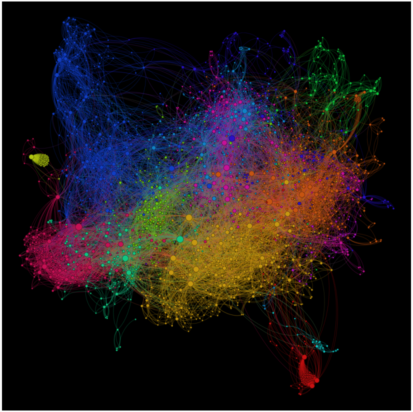
Application: Social media polarization

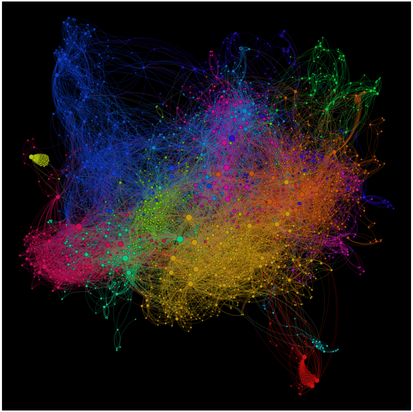


Questions: how to ideas flow between bloggers? Right now? Over time? Who's the most influential within a given party? In general?

http://allthingsgraphed.com/public/images/political-blogs-2004/left-right.png

Application: Analyzing code





Questions: which files import which ones? Which files are most used and should be optimized? What if two files import each other? If a file has a security vulnerability, how might it propagate?

http://allthingsgraphed.com/public/images/java-call-graph/cassandra-import-graph.png

Application of graphs

Other interesting questions:

- How does Google work?

Application of graphs

Other interesting questions:

- How does Google work?
- How do I solve Sudoku efficiently?

Application of graphs

Other interesting questions:

- How does Google work?
- How do I solve Sudoku efficiently?
- How do genes in my DNA influence and regulate each other?

Application of graphs

Other interesting questions:

- How does Google work?
- How do I solve Sudoku efficiently?
- How do genes in my DNA influence and regulate each other?
- How can I allocate registers to variables in a program?

Application of graphs

Other interesting questions:

- How does Google work?
- How do I solve Sudoku efficiently?
- How do genes in my DNA influence and regulate each other?
- How can I allocate registers to variables in a program?
- How similar/dissimilar are words? Based on spelling? Based on meaning?

Modeling problems with graph

How would you model the following using graphs? Decide what you think the vertices are and what the edges are:

- Maps (e.g. Google Maps)
- Web pages
- A running program
- Courses at UW
- A family tree

Modeling problems with graph

How would you model the following using graphs? Decide what you think the vertices are and what the edges are:

- Maps (e.g. Google Maps)
  Idea: vertices are intersections, edges are roads. How do we model traffic? Paths for cyclists vs cars? One-way roads?
- Web pages
- A running program
- Courses at UW
- A family tree

Modeling problems with graph

How would you model the following using graphs? Decide what you think the vertices are and what the edges are:

- Maps (e.g. Google Maps)
  Idea: vertices are intersections, edges are roads. How do we model traffic? Paths for cyclists vs cars? One-way roads?
- Web pages
  Idea: vertices are webpages, links are edges.
- A running program
  Idea: model each statement as vertices, and the next lines it can execute as edges.
- Courses at UW
- A family tree

Modeling problems with graph

How would you model the following using graphs? Decide what you think the vertices are and what the edges are:

- Maps (e.g. Google Maps)
  Idea: vertices are intersections, edges are roads. How do we model traffic? Paths for cyclists vs cars? One-way roads?
- Web pages
  Idea: vertices are webpages, links are edges.
- A running program
  Idea: model each statement as vertices, and the next lines it can execute as edges.
- Courses at UW
  Idea: model courses as vertices, and pre-requisites as edges.
- A family tree

Modeling problems with graph

How would you model the following using graphs? Decide what you think the vertices are and what the edges are:

- Maps (e.g. Google Maps)
  Idea: vertices are intersections, edges are roads. How do we model traffic? Paths for cyclists vs cars? One-way roads?
- Web pages
  Idea: vertices are webpages, links are edges.
- A running program
  Idea: model each statement as vertices, and the next lines it can execute as edges.
- Courses at UW
  Idea: model courses as vertices, and pre-requisites as edges.
- A family tree
  Idea: model people as vertices, and relations as edges. Or the other way around: model events like birth or divorce as vertices, and make people the edges connecting events?

Is there a “graph” ADT?

Question: is there a graph ADT?

Is there a “graph” ADT?

Question: is there a graph ADT?

Well, what operations belong to the ADT? Hmm, lots of ideas (`getEdge(v)`, `reachableFrom(...)`, `centrality(...)`, etc..)

Is there a “graph” ADT?

Question: is there a graph ADT?

Well, what operations belong to the ADT? Hmm, lots of ideas (`getEdge(v)`, `reachableFrom(...)`, `centrality(...)`, etc..)

Observation: It’s very unclear what the “standard operations” are

Instead, what we do is think about graphs abstractly, and think about which algorithms are relevant to the problems at hand.

Undirected graphs

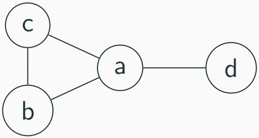
Undirected graph

In a **undirected graph**, edges have no direction: are two-way

Undirected graphs

Undirected graph

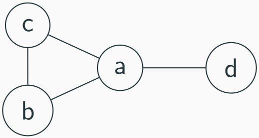
In a **undirected graph**, edges have no direction: are two-way



Undirected graphs

Undirected graph

In a **undirected graph**, edges have no direction: are two-way

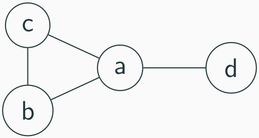


This means that $(x, y) \in E$ implies that $(y, x) \in E$. (Often, we treat these two pairs as equivalent and only include one).

Undirected graphs

Undirected graph

In a **undirected graph**, edges have no direction: are two-way



This means that $(x, y) \in E$ implies that $(y, x) \in E$. (Often, we treat these two pairs as equivalent and only include one).

Degree of a vertex

The **degree** of some vertex v is the number of edges containing that vertex.

So, the degree is the number of adjacent vertices.

Directed graphs

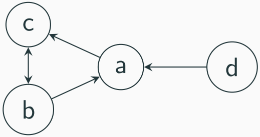
Directed graph

In a **directed graph**, edges *do* have a direction: are one-way

Directed graphs

Directed graph

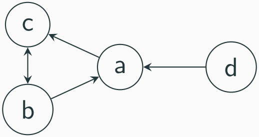
In a **directed graph**, edges *do* have a direction: are one-way



Directed graphs

Directed graph

In a **directed graph**, edges *do* have a direction: are one-way

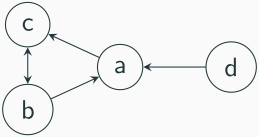


Now, (x, y) and (y, x) mean different things.

Directed graphs

Directed graph

In a **directed graph**, edges *do* have a direction: are one-way



Now, (x, y) and (y, x) mean different things.

In-degree of a vertex

The **in-degree** of v is the number of edges that point to v.

Out-degree of a vertex

The **out-degree** of v is the number of edges that start at v.

Self-loops and parallel edges

Self-loop

A **self-loop** is an edge that starts and ends at the same vertex.

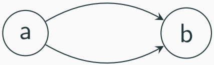
Self-loops and parallel edges

Self-loop

A **self-loop** is an edge that starts and ends at the same vertex.

Parallel edges

Two edges are **parallel** if they both start and end at the same vertices.



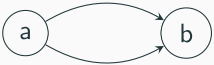
Self-loops and parallel edges

Self-loop

A **self-loop** is an edge that starts and ends at the same vertex.

Parallel edges

Two edges are **parallel** if they both start and end at the same vertices.



Whether we allow or disallow self-loops and parallel edges depends on what we're trying to model.

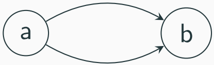
Self-loops and parallel edges

Self-loop

A **self-loop** is an edge that starts and ends at the same vertex.

Parallel edges

Two edges are **parallel** if they both start and end at the same vertices.



Whether we allow or disallow self-loops and parallel edges depends on what we're trying to model.

Simple graph

A graph with no self-loops and no parallel edges.

Some questions

Questions:

- In an undirected graph, is it possible to have a vertex with a degree of zero?
- In a directed graph, a vertex with an in-degree and out-degree of both zero?

Some questions

Questions:

- In an undirected graph, is it possible to have a vertex with a degree of zero?
  Yes.
- In a directed graph, a vertex with an in-degree and out-degree of both zero?
  Yes.