Code Fellows

# Class 20 + DSA 04

## Sorting, Props and State

seattle-javascript-401n14

# Holiday Break!

# Holiday Break

| | | | | | | |
|---|---|---|---|---|---|---|
| Monday<br>DEC 23 2019<br><br>Co-working | Tuesday<br>DEC 24 2019<br><br>Lecture | Wednesday<br>DEC 25 2019<br><br>Lab | Thursday<br>DEC 26 2019<br><br>Co-working | Friday<br>DEC 27 2019 | Saturday<br>DEC 28 2019<br><br>Lecture + Lab | Sunday<br>DEC 29 2019 |
| Monday<br>DEC 30 2019<br><br>Co-working | Tuesday<br>DEC 31 2019<br><br>Lecture | Wednesday<br>JAN 01 2020<br><br>Lab | Thursday<br>JAN 02 2020<br><br>Co-working | Friday<br>JAN 03 2020 | Saturday<br>JAN 04 2020<br><br>Lecture + Lab | Sunday<br>JAN 05 2020 |

# Lab 19 Review

# Code Challenge 19 Review

# Vocab Review!

# What is a CSS Preprocessor?

# What is a Sass Partial?

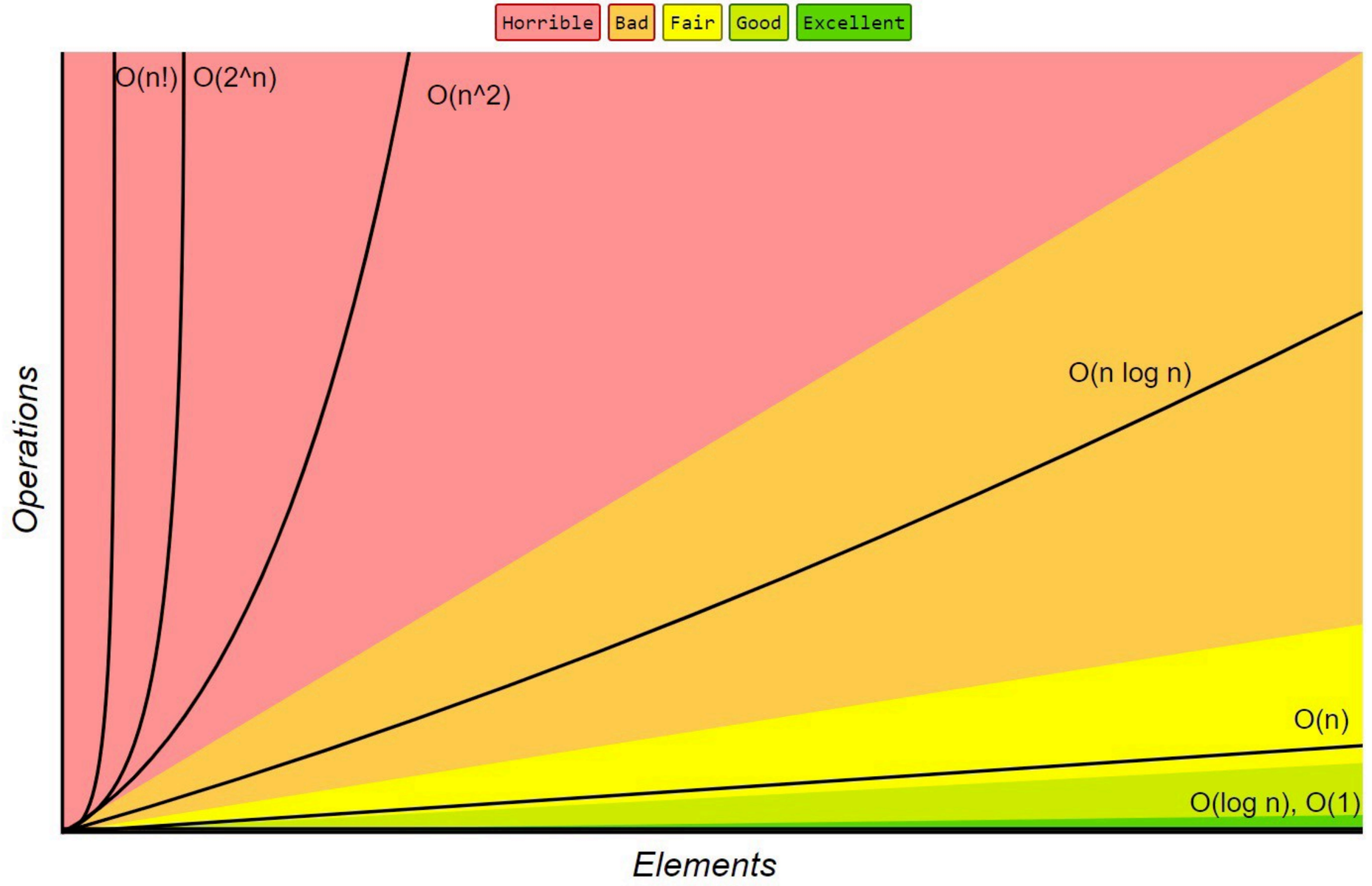
# What is a Mixin?

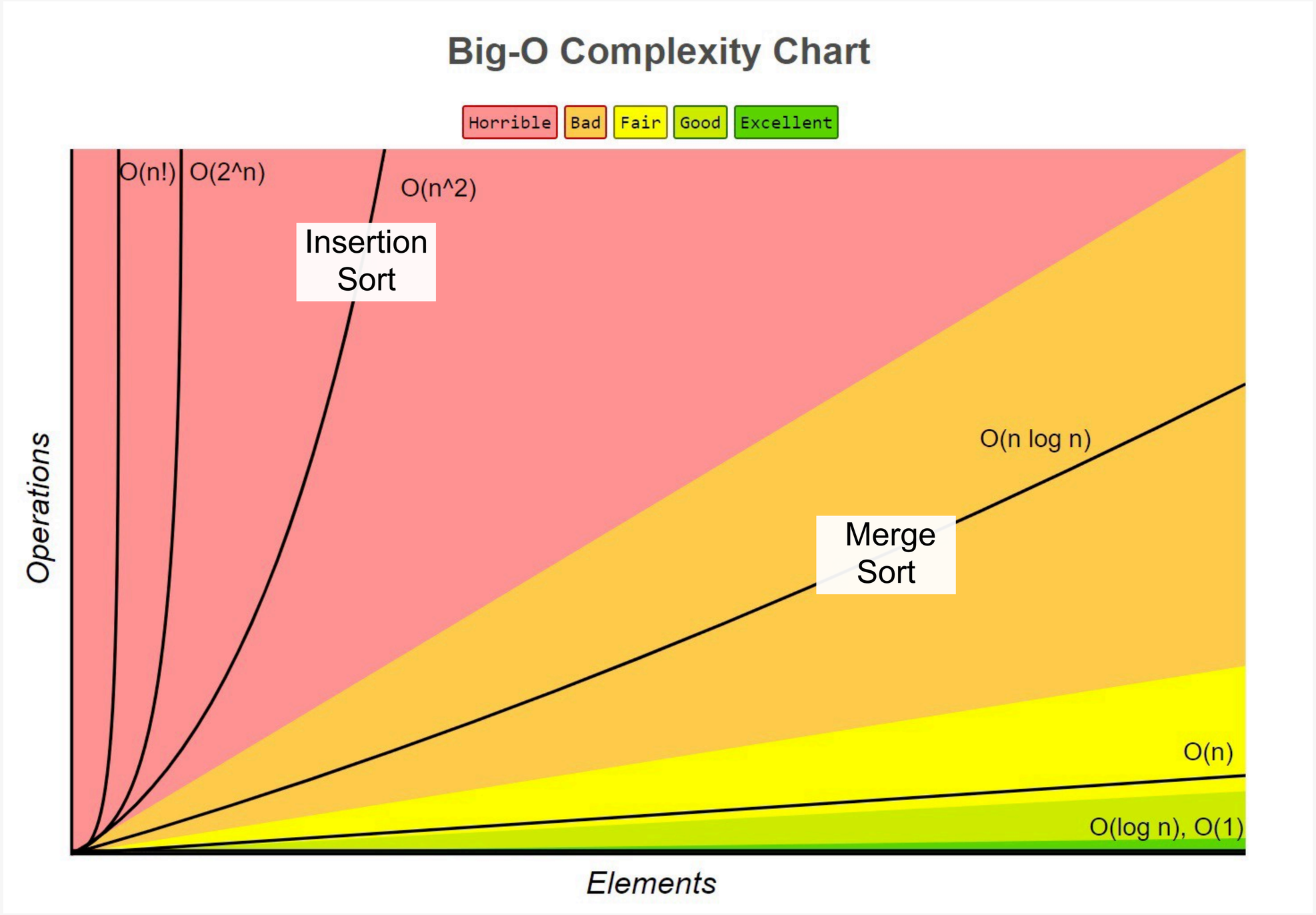
# What is snapshot testing?

# What is mount testing?

# What is big-o?

# Big-O Complexity Chart




Big-O Complexity Chart
Horrible
Bad
Fair
Good
Excellent
O(n!)
O(2^n)
O(n^2)
Insertion Sort
O(n log n)
Merge Sort
O(n)
O(log n), O(1)
Operations
Elements

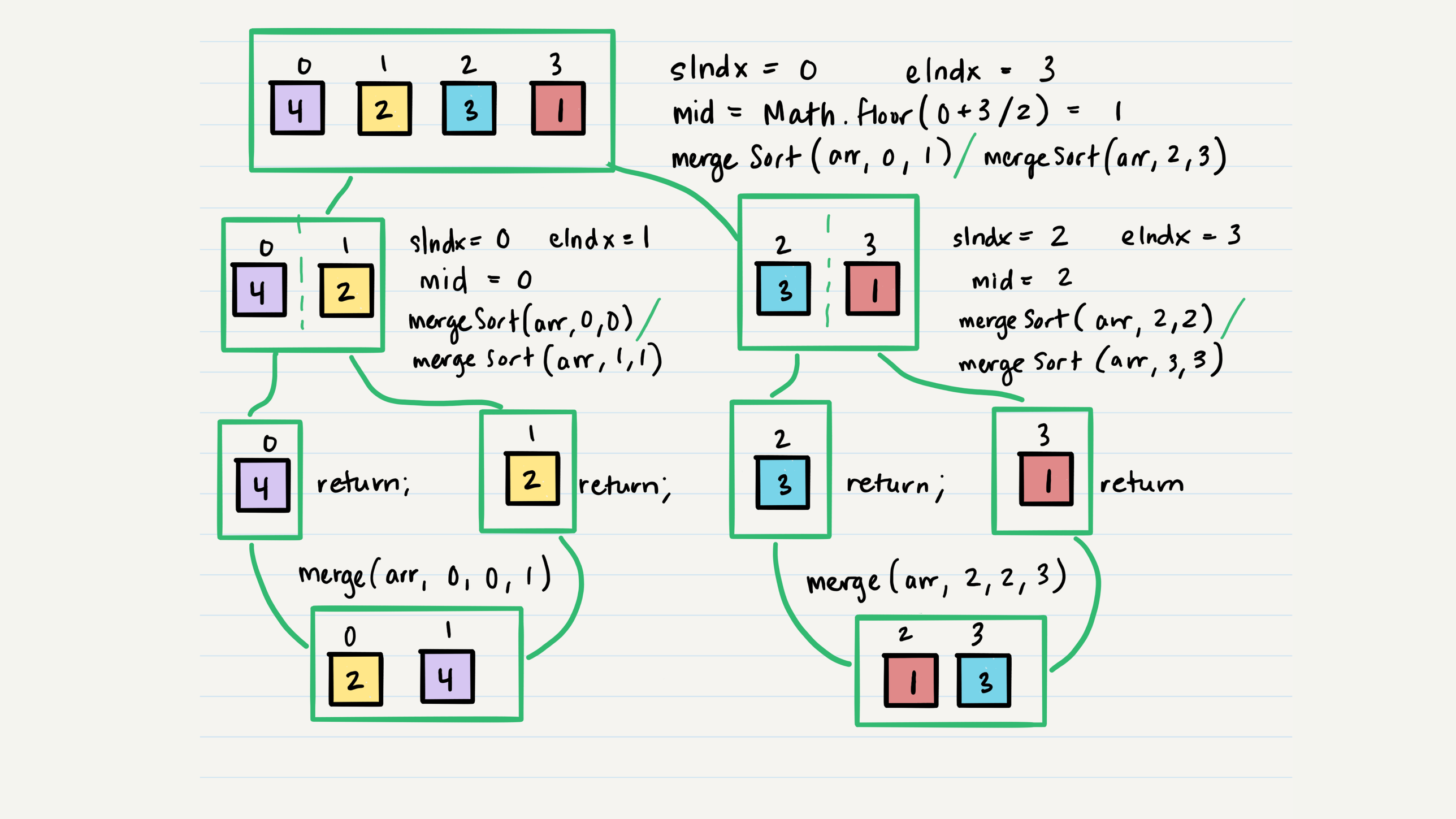
# Merge Sort

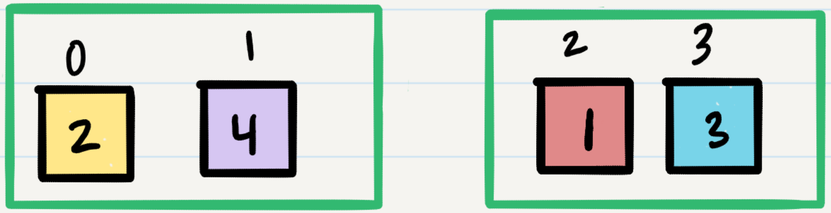
- Step 1 - If there is only one element in the array, it's already sorted so return
- Step 2 - Divide the array **recursively** into two halves until you can't divide
- Step 3 - Merge the smaller lists into a combined list in sorted order

```
function mergeSort(arr, sIndx, eIndx)
  if sIndx >= eIndx // array of size 1 or less
    return;
  let midpoint = Math.floor((sIndx + eIndx) / 2);
  mergeSort(arr, sIndx, midpoint);
  mergeSort(arr, midpoint, eIndx);
  merge(arr, sIndx, midpoint, eIndx);
```

# Merge Sort

```
function merge(arr, sIndx, mid, eIndx)
    let merged = [];
    let j = mid + 1;
    for i = 0; i <= mid; i++
        while arr[i] > arr[j] && j < arr.length
            merged.push(arr[j]);
            j++;
        merged.push(arr[i]);
    for i = 0; i < merged.length; i++
        arr[sIndx + i] = merged[i];
```


0 1 2 3
4 2 3 1
slndx = 0 elndx = 3
mid = Math.floor(0+3/2) = 1
mergeSort(arr, 0, 1) / mergeSort(arr, 2, 3)
0 1
4 2
slndx = 0 elndx = 1
mid = 0
mergeSort(arr, 0, 0) /
mergeSort(arr, 1, 1)
2 3
3 1
slndx = 2 elndx = 3
mid = 2
mergeSort(arr, 2, 2) /
mergeSort(arr, 3, 3)
0
4
return;
1
2
return;
2
3
return;
3
1
return
merge(arr, 0, 0, 1)
0 1
2 4
merge(arr, 2, 2, 3)
2 3
1 3







```
function merge(arr, sIndx, mid, eIndx)
  let merged = [];
  let j = mid + 1;
  for(i = 0; i <= mid; i++)
    while(arr[i] > arr[j] &&
          j < arr.length)
      merged.push(arr[j])
      j++;
    merged.push(arr[i])
```

merge(arr, 0, 1, 3)

merged = []; j = 2; i = 0

arr[0] > arr[2] ✓

2 > 1

merged.push(arr[2])

merged = [1] j++ = 3

arr[0] > arr[3] X

2 < 3

merged.push(arr[0])

merged = [1,2] j = 3 i = 1

arr[1] > arr[3] ✓

4 > 3

merged.push(arr[3])

merged = [1,2,3] j++ = 4

j < arr.length X

4 = 4

merged.push(arr[1])

merged = [1, 2, 3, 4]

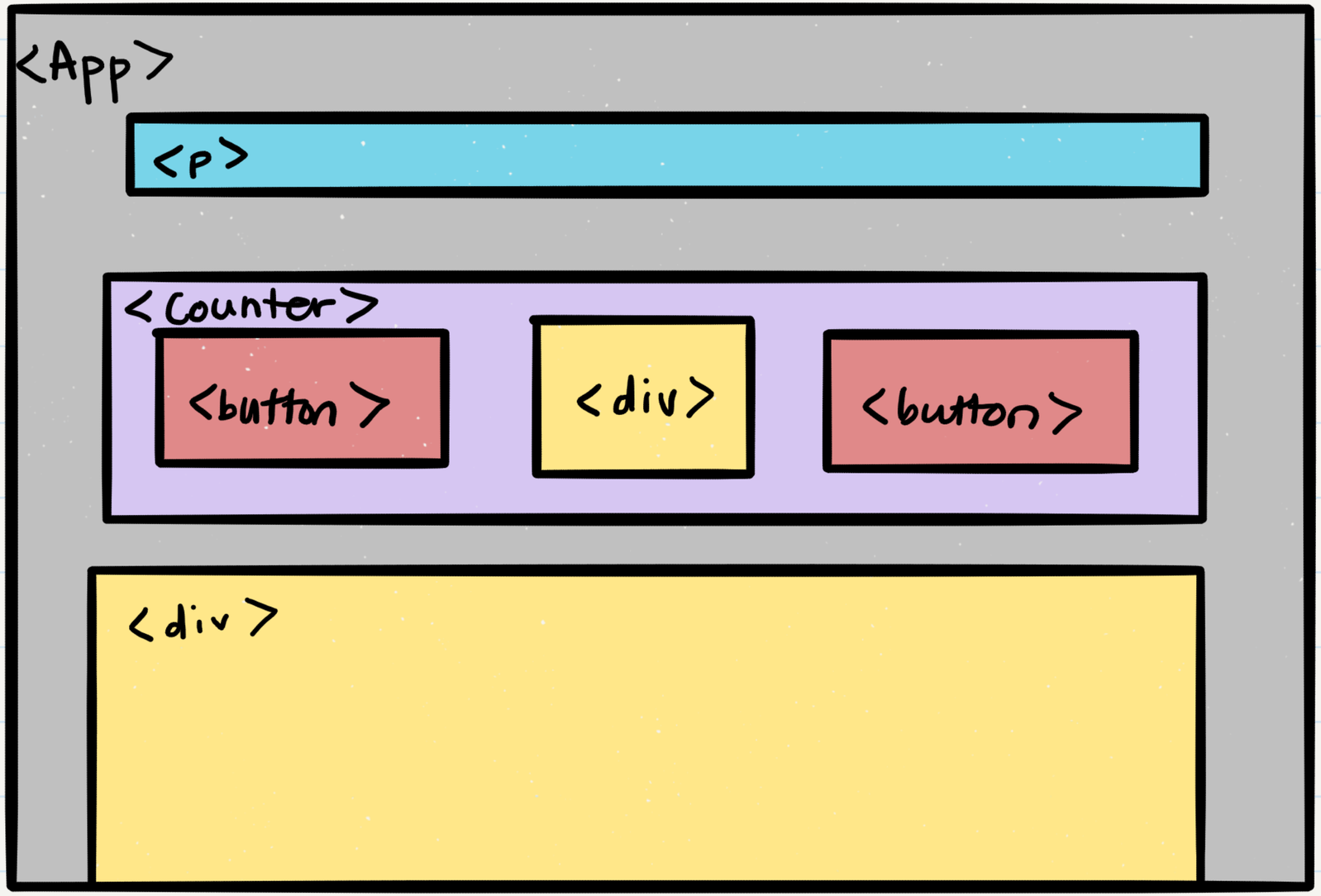
# Merge Sort

- Recursive `mergeSort` function with a `merge` function that has a `while` loop nested in a `for` loop
- Confusing structure = confusing complexity
- Use tally mark tactic for a small input
- A relationship that is hard to articulate is *usuall*y `logn` based
- Merge sort is `nlogn` - better than Insertion sort

# We'll resume sorting next class!

# Props and State (Again)

- **Props** are like function parameters, where the “functions” are either functional or class components
- **State** is like local variables which are meant to be dynamic
- React is structured like HTML where you have **parent** components containing **child** components


<App>
<p>
<Counter>
<button>

<button>

# Parent to Child

- If a parent has local state variables, they can pass them to child components using props
- Even better, **props can contain anything**!
  - Functions
  - Static variables
  - Strings
  - Any valid JavaScript object

# Child to Parent

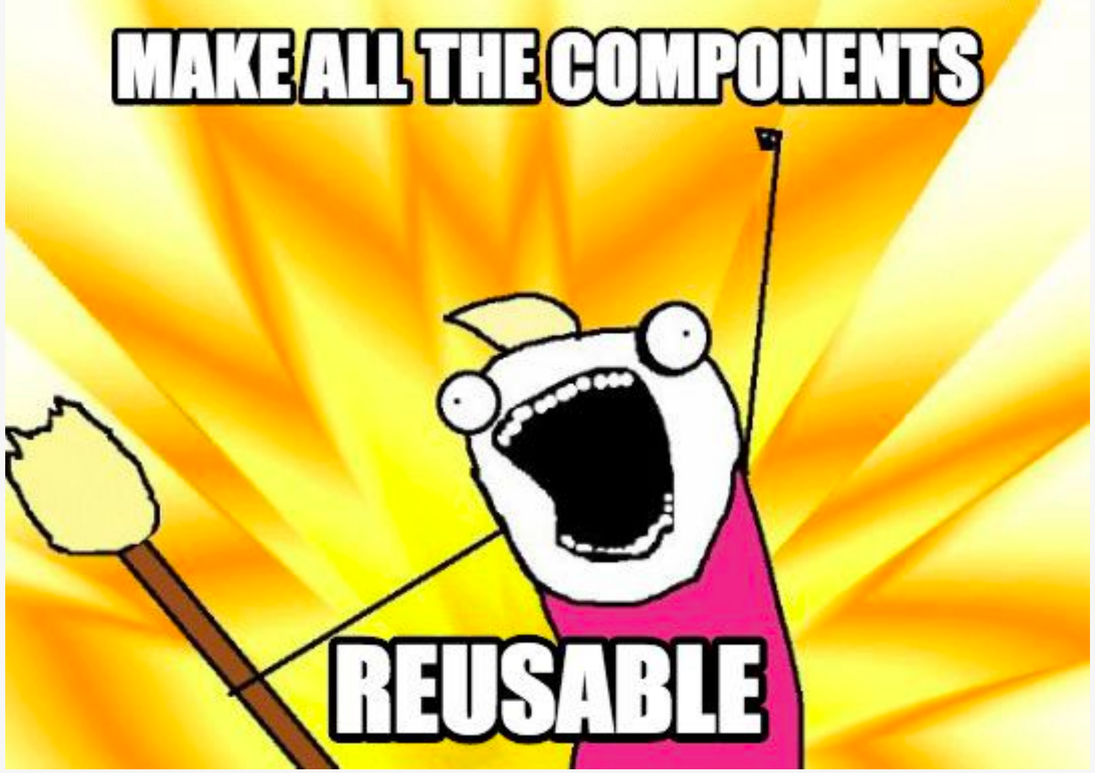
- Unfortunately, there's really no way for a child component to send data back up to the parent
- The exception is through callback / handler functions
- Mostly, the child will not be able to directly communicate with the parent
- We'll learn ways to get around this in the future
    - The child will write data to a shared space that the parent can access

# Why Do We Care?

- A big component of writing UI in React is that we can make pieces of our UI **generic** and **reusable**
- We usually view our child components as “**dumb**”
  - They don’t have a state
  - They don’t care about where they’re being used
  - They usually just take parameters and display them in a cool way
- Our parent components are “**smart**”
  - Maintain state
  - Have more complex logic

# Why Do We Care?

- By thinking in this way, we can ensure that we're breaking up our components correctly
- We should minimize "smart" components where we can, because they're usually not reusable
- We need to get used to building components that are completely independent from our application, so we can spread them around!
    - Counter you can add anywhere
    - Converter you can add anywhere
    - Form elements you can add anywhere



# Demo

Let’s play around with props and state to see how we can make smart and dumb components.

# Lab 20 Preview

# What's Next:

- Due by Midnight Tomorrow:
    - **Learning Journal 20**
- Due by Midnight Thursday:
    - **Code Challenge 20**
- Due by Saturday 9am:
    - **Lab 20**
    - **Reading Class 21**
- Next Class on Tuesday: **Class 21 - Routing and Component Composition**

# Questions?