Your Next Week

Saturday June 13	Sunday June 14	Monday June 15	Tuesday June 16
9:00 AM **— DUE Class 22 Lab** **— DUE Class 22 Code Challenge** **— DUE Class 23 Reading** — Class 23 — Interview Prep *MIDNIGHT* **— DUE Class 23 Learning Journal**	*MIDNIGHT* **— DUE Career: Coffee Networking Report** **— DUE Class 22-23 Feedback**		*6:30 PM* **— DUE Class 23 Lab** **— DUE Class 23 Code Challenge** **— DUE Class 24 Reading** — Class 24A
Wednesday June 17	**Thursday June 18**	**Friday June 19**	**Saturday June 20**
6:30 PM — Class 24B *MIDNIGHT* **— DUE Class 24 Learning Journal**	*6:30 PM* — Co-working		*9:00 AM* **— DUE Class 24 Lab** **— DUE Class 24 Mock Interviews** **— DUE Class 25 Reading** — Class 25 — Interview Prep *MIDNIGHT* **— DUE Class 25 Learning Journal**

What We've Covered

Module 01

Javascript Fundamentals and Data Models

C01 — Node Ecosystem, TDD, CI/CD
C02 — Classes, Inheritance, Functional Programming
C03 — Data Modeling & NoSQL Databases
C04 — Advanced Mongo/Mongoose
C05 — DSA: Linked Lists

Module 02

API Servers

C06 — HTTP and REST
C07 — Express
C08 — Express Routing & Connected API
C09 — API Server
C11 — DSA: Stacks and Queues

Module 03

Auth/Auth

C10 — Authentication
C12 — OAuth
C13 — Bearer Authorization
C14 — Access Control (ACL)
C15 — DSA: Trees

Module 04

Realtime

C16 — Event Driven Applications
C17 — TCP Server
C18 — Socket.io
C19 — Message Queues
C20 — Midterms Prep

Midterms

Module 05

React Basics

C21 — Component Based UI
C22 — React Testing and Deployment
C23 — Props and State
C24 — Routing and Component Composition
C25 — DSA: Sorting and HashTables

Module 06

Advanced React

C26 — Hooks API
C27 — Custom Hooks
C28 — Context API
C29 — Application State with Redux
C30 — DSA: Graphs

Module 07

Redux State Management

C31 — Combined Reducers
C32 — Asynchronous Actions
C33 — Additional Topics
C34 — React Native
C35 — DSA: Review

Module 08

UI Frameworks

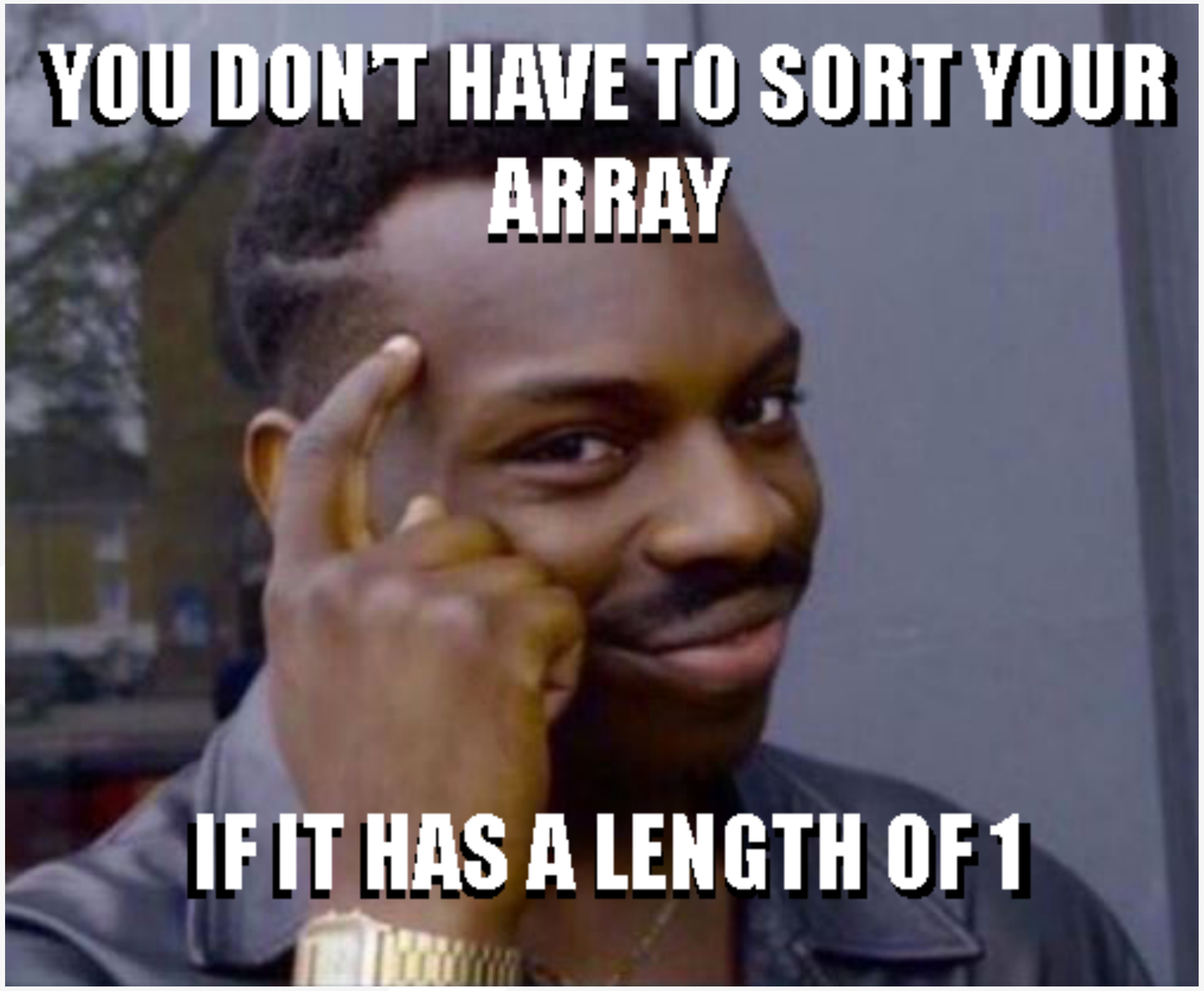
C36 — Gatsby and Next
C37 — JavaScript Frameworks
C38 — Finals Prep

Finals

Code Challenge 22
Review

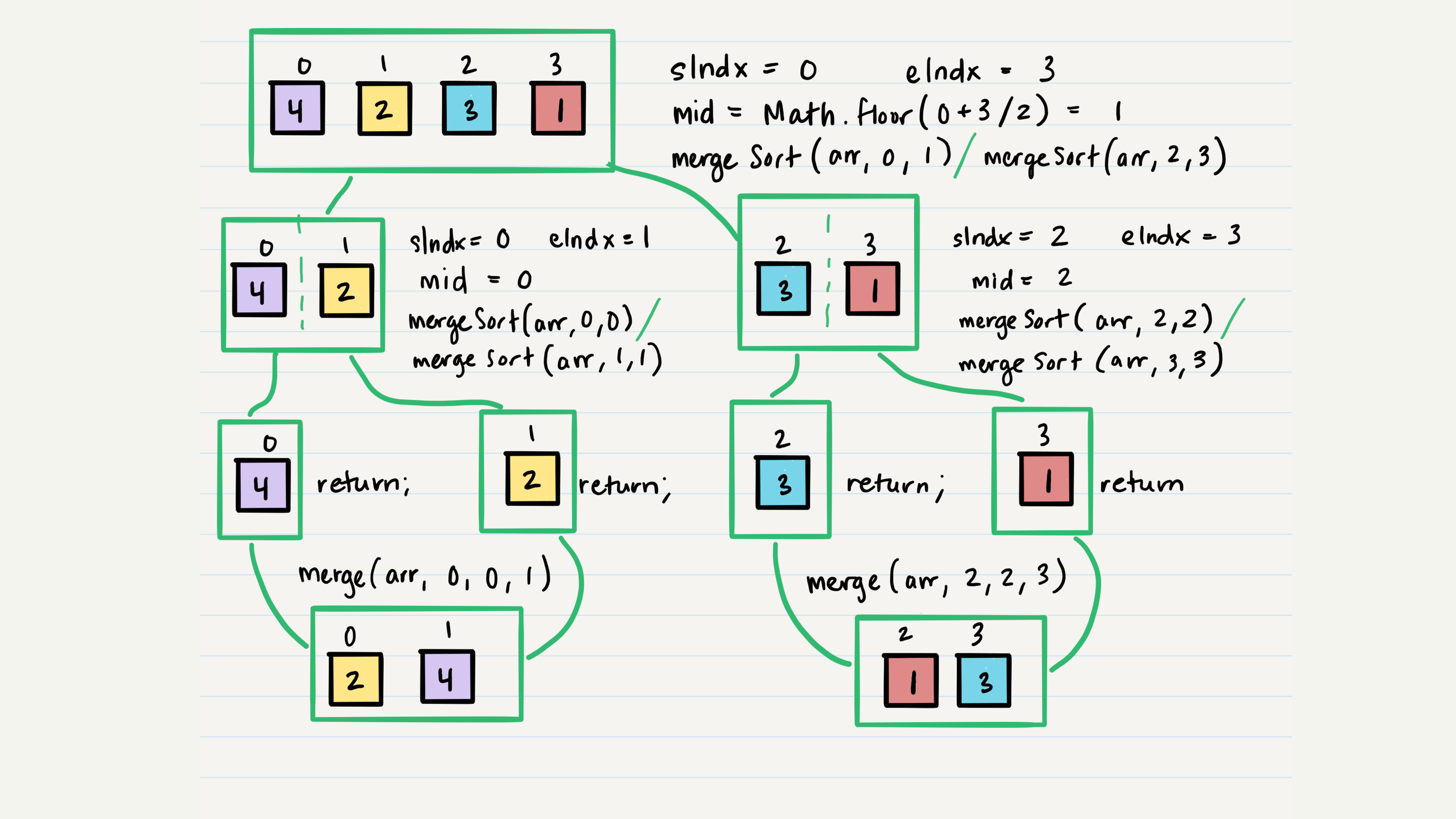
Merge Sort

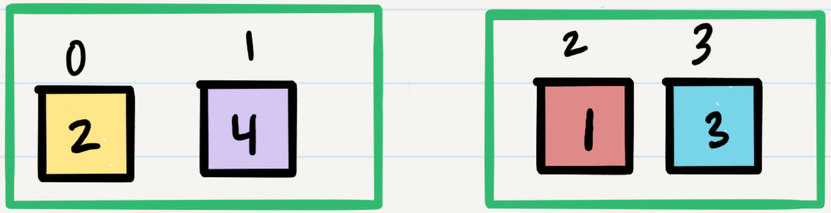
- Step 1 - If there’s only one element in the array, it’s already sorted, so return
- Step 2 - Divide the array recursively into two halves until you can’t divide further
- Step 3 - Merge the smaller lists into a combined list in sorted order



```
function merge(arr, sIndx, eIndx) {
    let merged = [];

    let mid = Math.floor(eIndx - sIndx);

    for (let i = 0; i <= mid; i++) {
        while (arr[i] > arr[j] && j <= eIndx) {
            merged.push(arr[j]);
            j++;
        }

        merged.push(arr[i]);
    }

    for (let i = 0; i < merged.length; i++) {
        arr[sIndx + i] = merged[i];
    }
}
```

```
function mergeSort(arr, sIndx = 0, eIndx = arr.length - 1) {
    let mid = Math.floor(eIndx - sIndx);

    mergeSort(arr, sIndx, mid);
    mergeSort(arr, mid + 1, eIndx);

    merge(arr, sIndx, eIndx);
}
```


0 1 2 3
4 2 3 1
sIndx = 0 eIndx = 3
mid = Math.floor(0+3/2) = 1
mergeSort(arr, 0, 1) / mergeSort(arr, 2, 3)
0 1
4 2
sIndx = 0 eIndx = 1
mid = 0
mergeSort(arr, 0, 0) /
mergeSort(arr, 1, 1)
2 3
3 1
sIndx = 2 eIndx = 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, slndx, mid, elndx)
  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

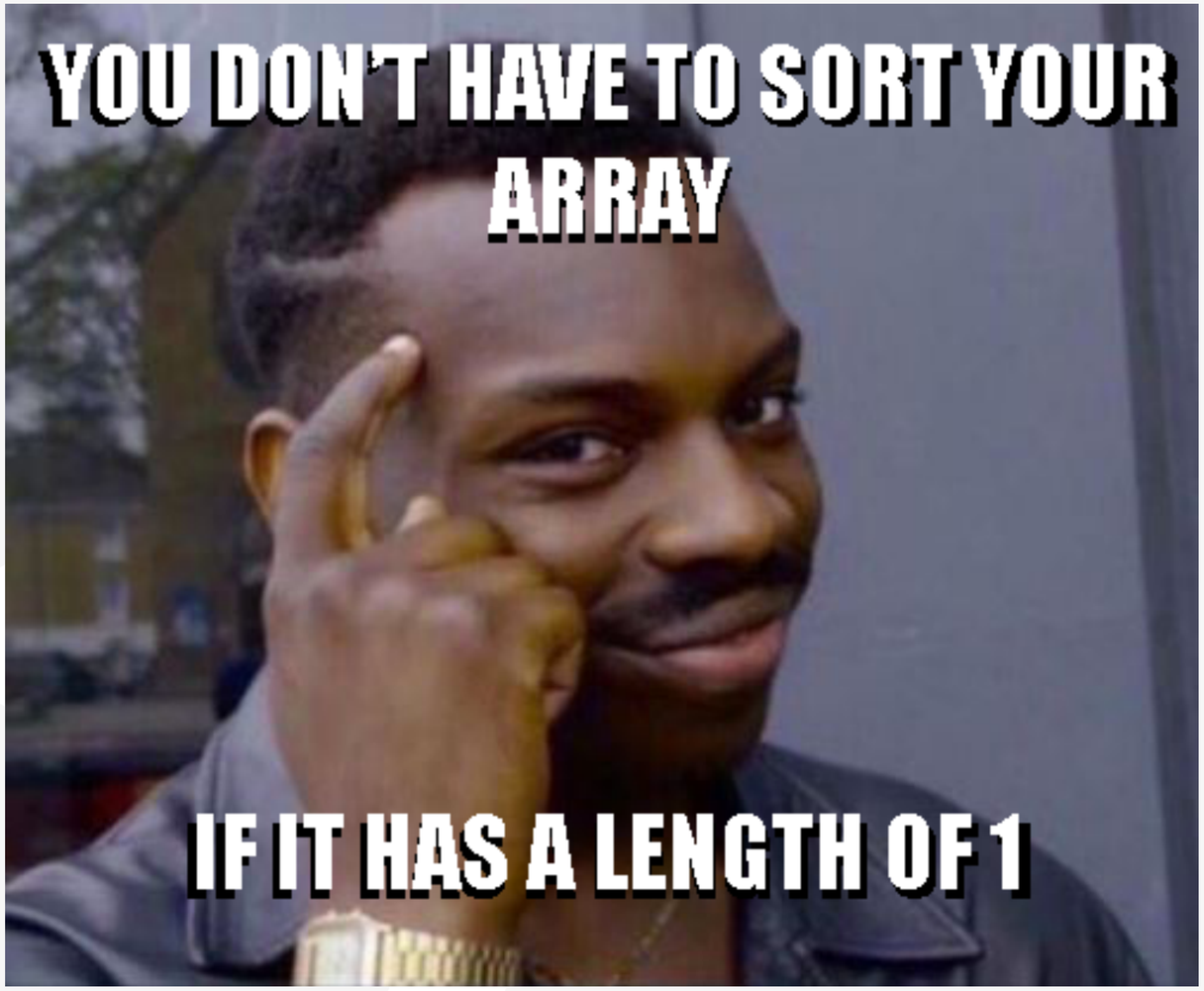
4 = 4

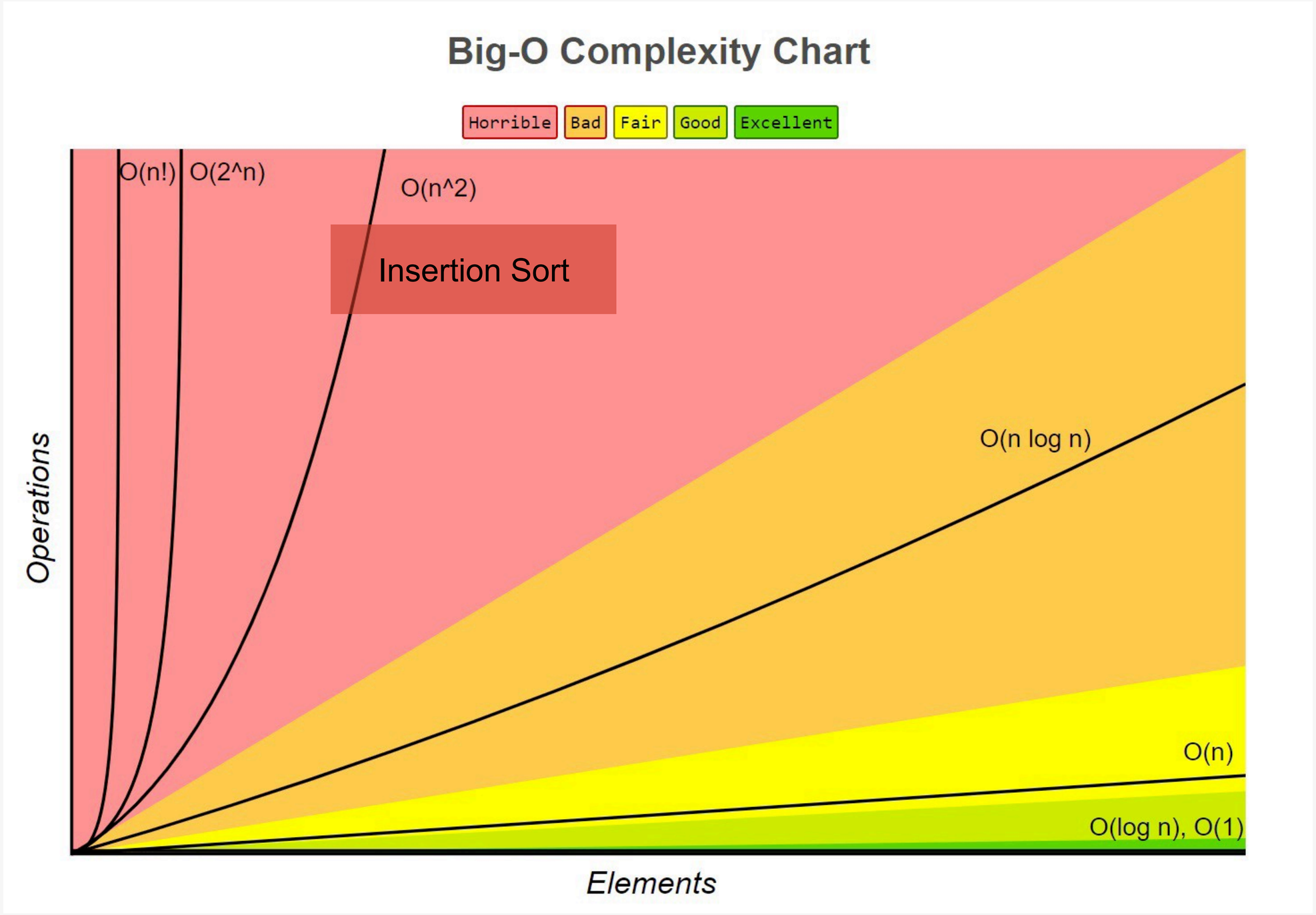
merged.push(arr[1])

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`




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

Lab 22 Review

Class 23

Props and State

seattle-javascript-401n16

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

Dumb and Smart Components

- 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

Lab 23 Overview