Your Next Week

Tuesday June 16	Wednesday June 17	Thursday June 18	Friday June 19
6:30 PM **— DUE Class 23 Lab** **— DUE Class 23 Code Challenge** **— DUE Class 24 Reading** — Class 24A	*6:30 PM* — Class 24B *MIDNIGHT* **— DUE Class 24 Learning Journal**	*6:30 PM* — Co-working	
Saturday June 20	**Sunday June 21**	**Monday June 22**	**Tuesday June 23**
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**	*MIDNIGHT* **— DUE CCW #2 Prep - Behavioral Questions** **— DUE CCW #2 Prep - Mock Interviews** **— DUE CCW #2 Prep - Star Methodology** **— DUE CCW #2 Prep - Winning the Interview** **— DUE Class 24-25 Feedback**	*6:30 PM* — Career Coaching Workshop 2A	*6:30 PM* **— DUE Class 25 Lab** **— DUE Class 26 Reading** — Class 26A

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

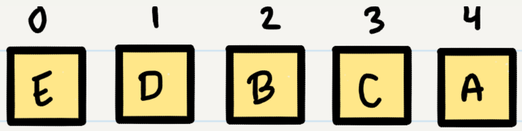
C36 — Gatsby and Next
C37 — JavaScript Frameworks
C38 — Finals Prep

Finals

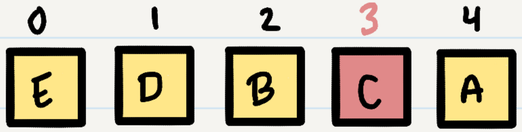
Code Challenge 23
Review

Quick Sort

- Step 1 - Pick an item in the array as the **pivot**
- Step 2 - Swap the pivot with the item at the end of the array
- Step 3 - Find the **first item larger** than the pivot, searching left to right
- Step 4 - Find the **first item smaller** than the pivot, searching right to left
- Step 5 - Swap these **two values**
- Repeat Step 3-5, until **first item larger** is at a greater index than **first item smaller**, or out of bounds
- Step 6 - Swap the last found **item larger than pivot** with the **pivot**
- Step 7 - The pivot should now be in the correct position. Recursively repeat this with the left half of the array before the pivot, and the right half of the array after the pivot



Step 1: Pick a Pivot



pivot = 3

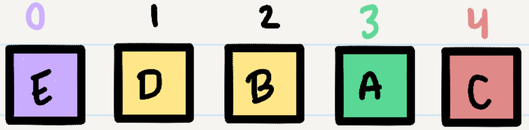
Step 2: Swap the pivot with the last item in the array

pivot = 4

Step 3: Find the first item larger than the pivot, left → right

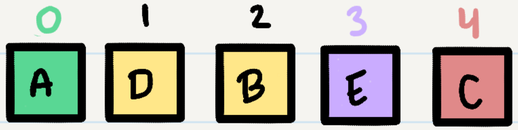
fil = 0 pivot = 4

Step 4: Find the first item smaller than the pivot, right → left



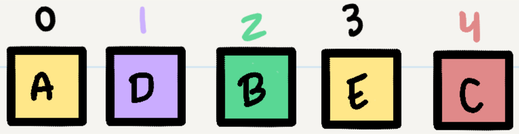
fil = 0 pivot = 4 fis = 3

Step 5: Swap the two values

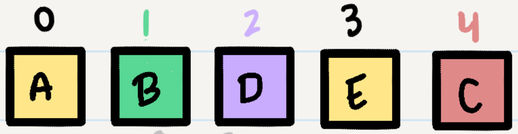


pivot = 4

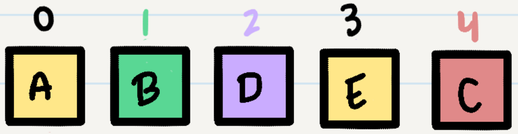
Repeat steps 3 - 5 until fil > fis



pivot = 4 fil = 1 fis = 2



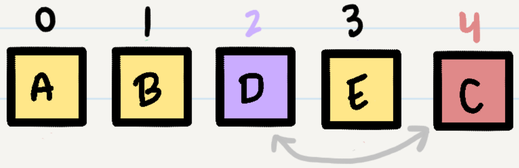
swap!

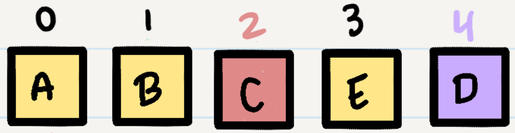


now, fil = 2 fis = 1

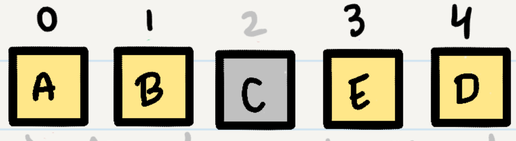
fil > fis!
2 > 1

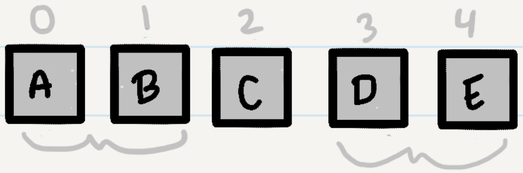
Step 6: Swap the last found item larger than pivot with the pivot

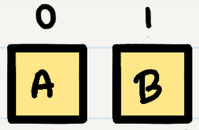


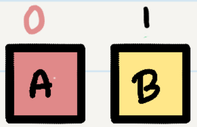


Step 7: The pivot should now be in the correct position. Recursively repeat with each partition

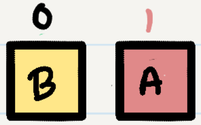




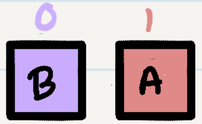




pivot = 0



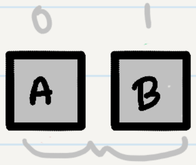
pivot = 1

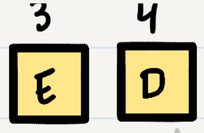


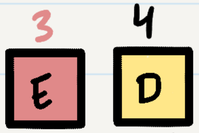
pivot = 1

fil = 0 fis = -1

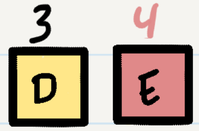
0 > -1, swap fil & pivot



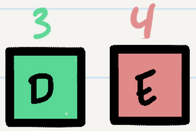




pivot = 3

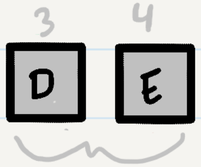


pivot = 4



pivot = 4

fil = 5 fis = 3



```
function quickSort(arr, sIndx, eIndx) {
    let pivot;

    if (sIndx < eIndx) {
        pivot = partition(arr, sIndx, eIndx);
        quickSort(arr, sIndx, pivot - 1);
        quickSort(arr, pivot + 1, eIndx);
    }
}
```

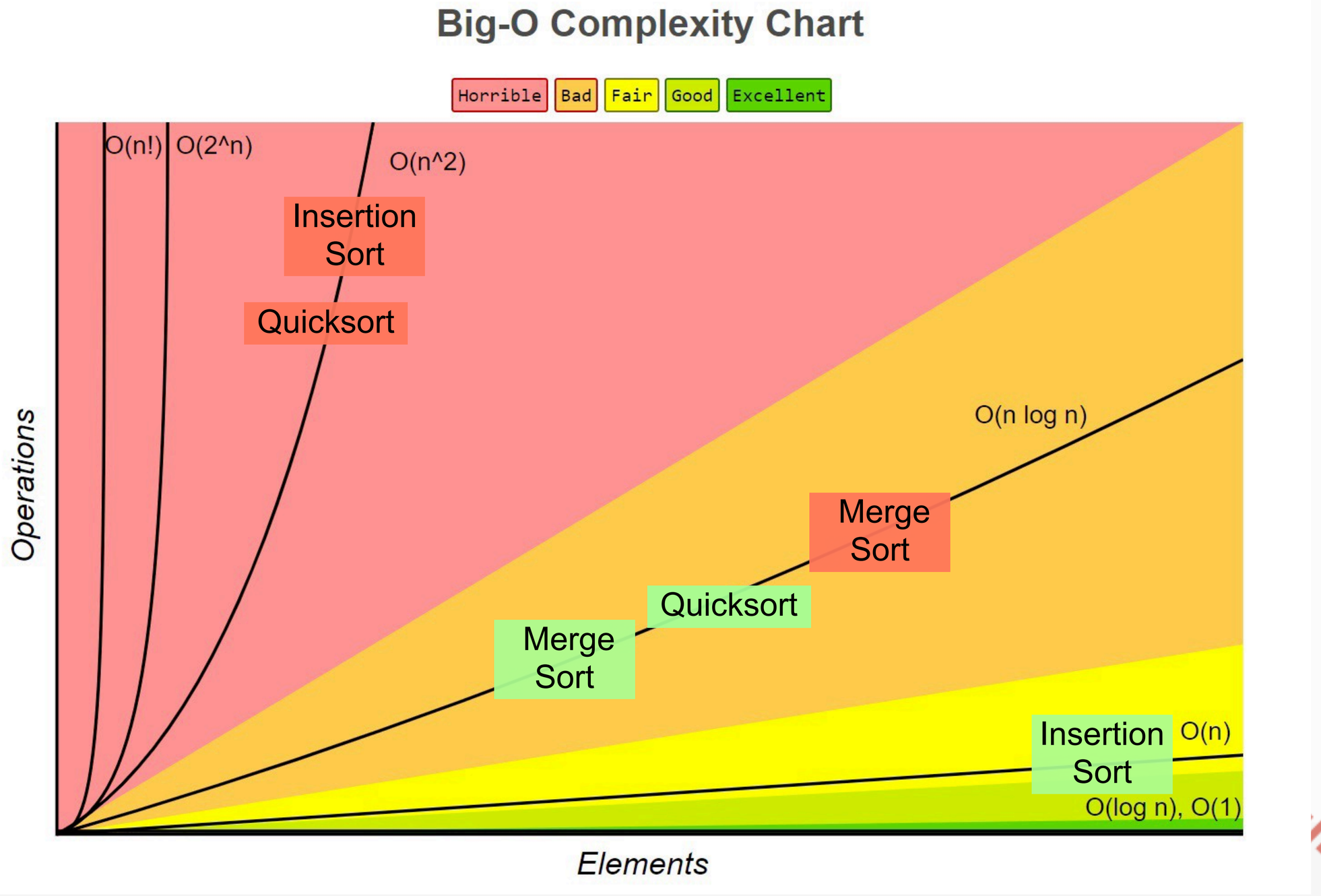
```
function partition(arr, sIndx, eIndx) {
    let pivot = eIndx;
    let fil = sIndx;
    let fis = eIndx;

    while (pivot != fil && fil <= eIndx && fis >= sIndx) {
        for (let i = 0; i < eIndx, i++)
            if (arr[i] > arr[pivot]) {
                fil = i;
                break;
            }

        for (let i = eIndx; i >= 0; i--) {
            if (arr[i] < arr[pivot]) {
                fis = i;
                break;
            }
        }

        if (fil < fis) swap(arr[fil], arr[fis]);
        else if (fil > fis) swap(arr[fil], arr[pivot]);
        pivot = fil;
    }

    return pivot;
}
```

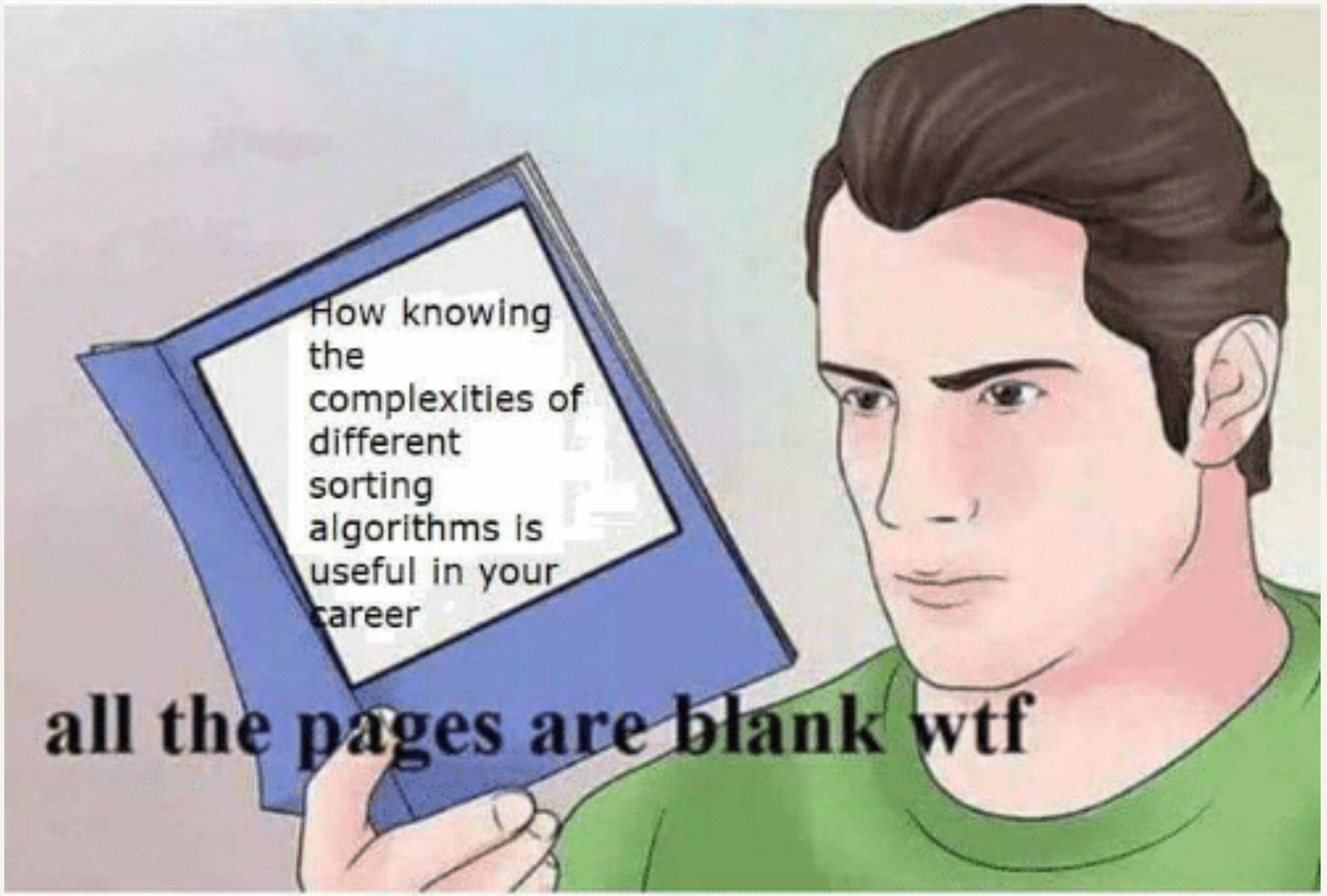
Quick Sort Complexity

- Depends on your pivot!
  - You can pick the ending index as pivot
  - You can randomly pick a pivot
  - You can randomly pick three array elements, compare values, and use middle value as pivot
- Best performance when pivot is the median value
- Worst case is $O(n^2)$, best case is $O(nlogn)$
- Why is this better than Insertion Sort, whose best case is $O(n)$?
  - Insertion sort *almost always* gets worse as n increases
  - Quicksort is *usually* $O(nlogn)$; it only changes to $O(n^2)$ when we pick a bad pivot, and that's actually pretty rare, as most pivots will result in $O(nlogn)$

Quicksort Complexity

- The complexity of Quicksort depends on how you pick the pivot
- Best performance when the pivot is the median value of the array
    - Some people always choose the right-most element
    - Some people randomly pick an element
    - Some people pick three random elements, find the middle of those three, and use that
- Worst case is $O(n^2)$, best case is $O(n \log n)$
- Why is this better than Insertion Sort, whose best case is $O(n)$?
    - Insertion sort *almost always* gets worse as n increases
    - Quicksort is *usually* $O(n \log n)$; it only changes to $O(n^2)$ when we pick a bad pivot, and that's actually pretty rare, as most pivots will result in $O(n \log n)$


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


How knowing the complexities of different sorting algorithms is useful in your career
all the pages are blank wtf

Lab 23 Review

Class 24

Routing and Component Composition

seattle-javascript-401n16

Props (yet again)

- Props are like parameters - we can define as many as we want
- There are default parameters exposed as well!
  - The main on is props.children
  - Very powerful and can make our components even more useful

Routing

- React so far has been running on just `index.html`
- How do we have new “pages”?
- We don’t!
- We tell React to render different components in the `<body>` tag of `index.html`
- We do this via a package called React Router (`npm install react-router-dom`)

Lab 24 Overview