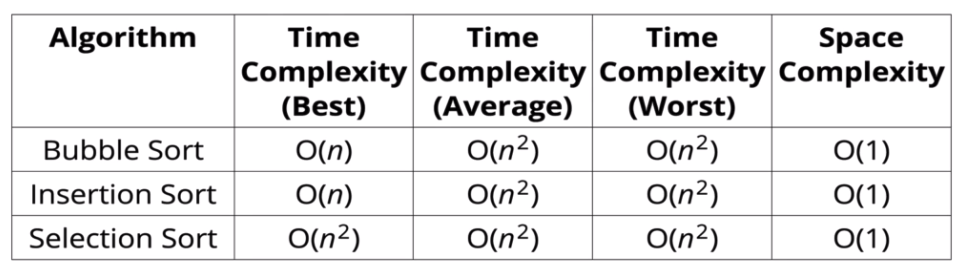
Algos Notes

Problem Solving Patterns

* Frequency Counter
  + Anytime you have multiple pieces of data and you need to compare them
  + In particular, if you need to see if they consist of the **same individual pieces** 
    - Anagrams, numbers vs numbers squared, if numbers consist of same digits just in a different order, etc
* Multiple Pointers
  + Creating pointers or values that correspond to an index or position and move towards the beginning, end, or middle based on a certain condition
  + Very efficient for solving problems with minimal space complexity as well
* Sliding Window
  + This pattern involves creating a **window** which can either be an array or number from one position to another
  + Depending on certain condition, the window either increases or closes (and a new window is created)
  + Very useful for keeping track of a subset of data in an array/string etc
* Divide and Conquer
  + This pattern involves dividing a data set into smaller chunks and then repeating a process with a subset of data
  + This pattern can tremendously decrease time complexity
  + Think finding a number in a sorted array, can look at the middle number, see if its less than or greater than, and only look at the half where the number can be, and continue to do that until the answer is found

**Sorting Algorithms**



Bubble Sort (O(n^2) worst case, O(n) best case)

* Start looping with a variable called i from the end of the array towards the beginning
* Start an inner loop with a variable called j from the beginning until i-1
* If arr[j] is greater than arr[j+1], swap those two values
* Return the sorted array
* Can add noSwaps variable to check if any swaps were made this pass, and if not, then break out of the outer loop b/c we know we’re already sorted

Selection Sort (O(n^2) always, only good if you want to reduce you number of swaps)

* Similar to bubble sort, but instead of placing largest values into sorted position, it places smaller values into sorted position
  + Store he first element as the smallest value you’ve seen so far
  + Compare this item to the next item in the array until you fins a smaller number
  + If a smaller number is found, designate that smaller number to be the new “min” and continue until the end of the array
  + If the “min” is not the value (index) you initially began with, sway the two values
  + Repeat this with the next element until the array is sorted

Insertion Sort (O(n^2) worst case, O(n) if data is almost all sorted)

* Builds up the sort by gradually creating a larger left half which is always sorted
  + Start by picking the second element in the array
  + Now compare the second element with the one before it and swap if necessary
  + Continue to the next elements and if it is in the incorrect order, iterate through the sorted portion (i.e. the left side) to place the element in the correct place
  + Repeat until array is sorted
* Works really well when data is coming in live and you need to sort each item as it comes in

Merge Sort (O(n log(n)) time always, O(n) space)

* First split up array into two arrays, continue to do that until you have 1 or 0 element arrays
  + Because 1 or 0 element arrays are always sorted!
* Then write a function that sorts two (already sorted) arrays, and call that until all of the small arrays have been merged into one sorted array!

Quick Sort (O(nlog(n)) avg and best, O(n^2) worst [time], O(nlog(n)) [space])

* Like merge sort, exploits the face that arrays of 0 or 1 element are always sorted
* Works by selecting one element (called the pivot) and finding the index where the pivot should end up in a sorted array
* Once the pivot is positioned appropriately, quick sort can be applied on either side of the pivot
* Pivot Helper
  + In order to implement merge sortm it’s useful to first implement a function responsible for arranging elements in an array on either side of a pivot
  + Given an array, this helper function should designate an element at the pivot
  + It should then rearrange elements in the array so that all values less than the pivot are moved to the left of the pivot, and all values greater than the picot are moved to the right of the pivot
  + The order of elements on either side of the pivot doesn’t matter!
  + The helper should do this in place, not create a new array
  + When complete, the helper should return the index of the pivot
* Picking a pivot
  + The runtime of quick sort depends in part on how one selects the pivot
  + Ideally, the pivot should be chosen so that it’s roughly the median value in the data set you’re sorting
  + For simplicity, we’ll always choose the pivot to be the first element (we’ll talk about the consequences of that later)
* Pivot Pseudocode
  + Accepts 3 arguments: array, start index, end index
  + Grab the pivot from the start of the array
  + Store the current pivot index in a variable (this will keep track of where the pivot will end up)
  + Loop through the array from the start until the end
    - If the pivot is greater than the current element, increment the pivot index variable and the swap the current element with the element at the pivot index
  + Swap the starting element (ie the pivot) with the pivot index
  + Return the pivot index
* Quicksort Pseudocode
  + Call the pivot helper on the array
  + When the helper returns to you the updated pivot index, recursively call the pivot helper on the subarray to the left of that index, and the subarray to the right of that index
  + Your base case occurs when you consider a subarray with less than 2 elements (where you start and end on you array are the same index)
  + Return the array
* Time & space complexity
  + Worst case would be if the pivot item you choose is always the minimum or maximum element in the array, b/c you’re not removing any items from pivot when you call it, for O(n^2)
  + This is why choosing the first element is a bad idea, in case the array os already sorted
  + Way to avoid this is to choose a random element or the middle number as your pivot, to avoid that possibility of occurring

Radix Sort

* Not a direct comparison algorithm like all of the other ones we’ve do so far!
  + All comparison sorts have a min time of nlog(n)
* This is a special sorting algorithm that works on lists of numbers
* Radix sort exploits the fact that information about the size of the number is encoded in the number of digits (4 digit number > 3 digit number always!)
* Need to create some helper functions first to get this to work
  + getDigit(num,place) - returns the digit in num at the given place (starting at ones digit)
  + digitCount(num) - returns the number of digits in num
  + mostDigits(nums) - given an array of nums, returns the max number of digits in the list
* Radix Sort Pseudocode
  + Figure out how many digits the largest number has
  + Loop from k = 0 up to the largest number of digits
  + For each iteration of the loop
    - Create buckets for each digit (0 to 9)
    - Place each number in the corresponding bucket based on its kth digit
  + Replace our existing array with values in our buckets, starting with 0 and going up to 9
  + Return the list at the end
* Time & Space Complexity (n = length of array, k = num of digits (average))
  + Time complexity (always) = O(n\*k)
  + Space complexity = O(n+k)

Singly Linked Lists

* Big O
  + Insertion = O(1) 🡪 (NOTE: not true for arrays! O(n) for shift, O(1) for push)
  + Removal = O(1) or O(n), depending on where the removal is
  + Access = O(n) 🡪 (NOTE: Arrays are better for this)
  + Search = O(n)
  + Singly Linked List beat Arrays for insertion and deletion at beginning!
    - But lose out on random access

Doubly Linked Lists

* Big O
  + Insertion = O(1) 🡪 (NOTE: not true for arrays! O(n) for shift, O(1) for push)
  + Removal = O(1) 🡺 NOTE: this is better than singly LL because you can go from either direction!
  + Access = O(n) 🡪 (NOTE: Arrays are better for this)
  + Search = O(n) 🡪 It’s technically O(n/2) but that’s still O(n)
  + Doubly LL are better if you want to move backwards through the list and are better for searching.
    - However, they take up more memory than a Singly LL b/c of the extra pointer

Stacks

* Used to handle function invocation (like call stack), operations like undo/redo, and routing (webpages you have visited, go back/forward)
* Can use push and pop from JS arrays for a stack, but can also use a linked list
* Big O
  + Insertion = O(1)
  + Removal = O(1)
    - Two above are really the only ones you’ll use in a stack, and it’s quick!
  + Access = O(n)
  + Search = O(n)

Queues

* Used to handle anything where you have an order and need to maintain that order. Like processing tasks and for more complex data structures
* Can use push and shift from JS arrays for a queue (not efficient), but can also use a linked list
* Big O
  + Insertion = O(1)
  + Removal = O(1)
    - Two above are really the only ones you’ll use in a queue, and it’s quick!
  + Access = O(n)
  + Search = O(n)

Binary Search Trees

* Big O
  + Insertion = O(log n) best and average case, O(n) worst case
  + Search = O(log n) best and average case, O(n) worst case

Tree Traversal

* Time complexity for BFS vs. DFS is the same 🡪 O(n)
* BFS
  + If tree is really fleshed out (aka wide tree), the there are more nodes to keep track of vs. DFS 🡪 larger space complexity
  + If the tree is narrow (looks like a linked list), the BFS is better for space
* Use cases for DFS
  + InOrder means you will get all of your nodes in order, so if you have a BST and you want a sorted list of your nodes, use InOrder
  + PreOrder can be used to “export” a tree structure so that it is easily reconstructed or copied

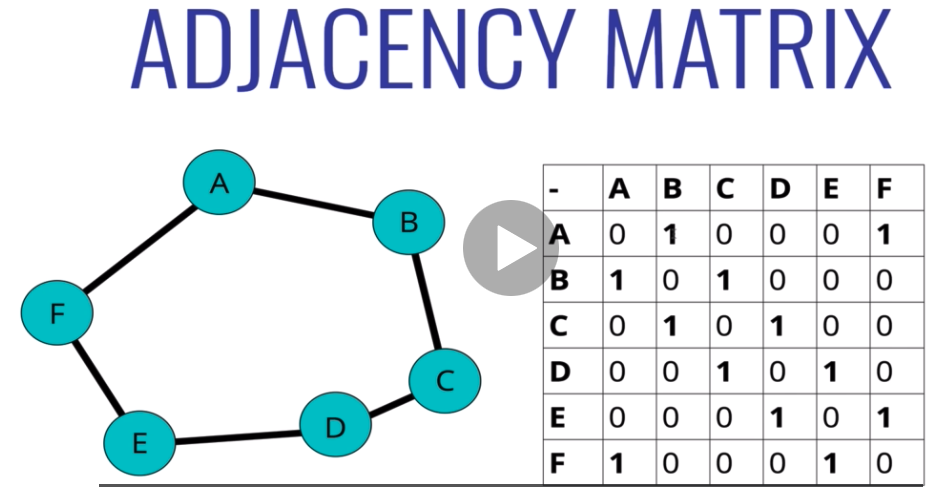
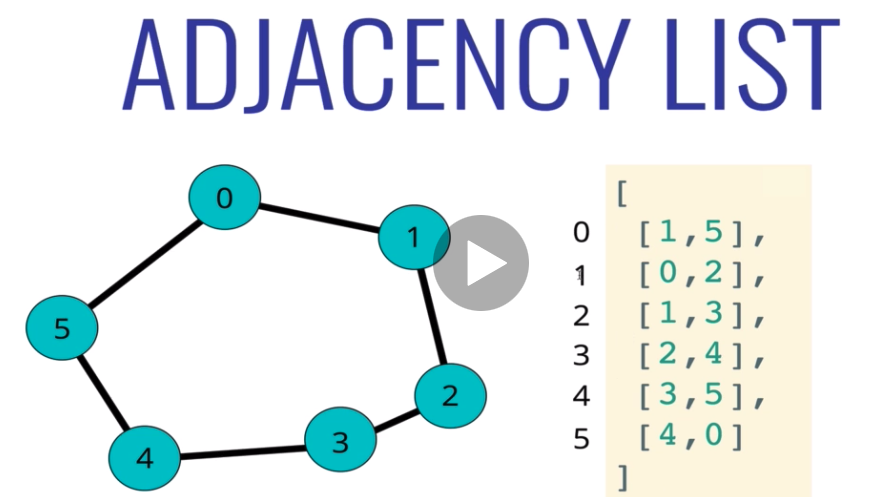
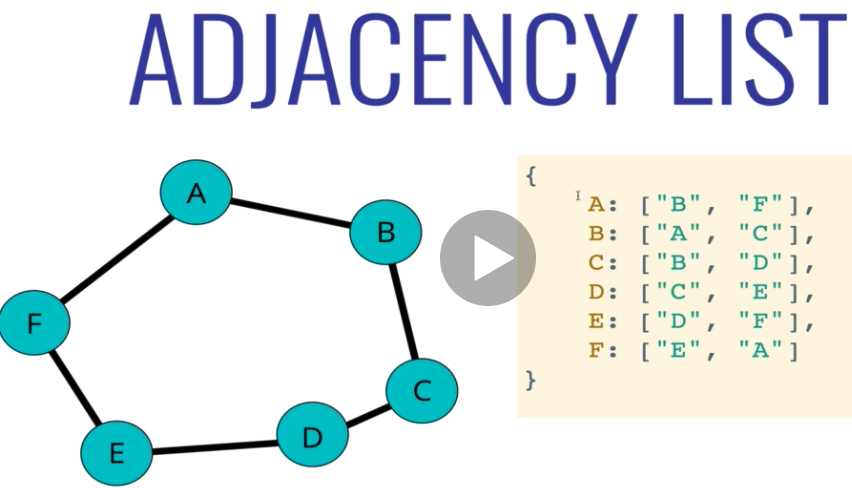
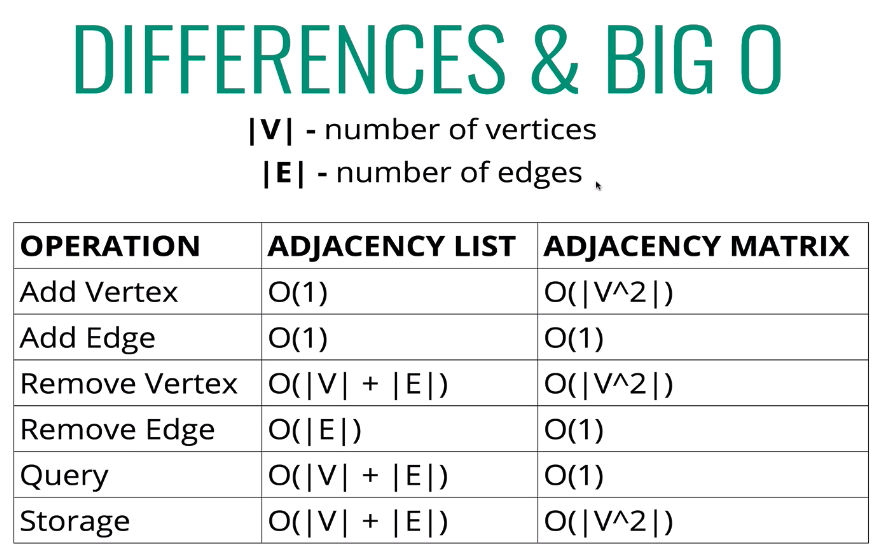
Binary Heaps

* Binary heaps can be stored as arrays in JS with a little math
  + For any parent node at index n, the left child is stored at 2n+1 and the right child is stored at 2n+2
  + For any child node at index n, the parent is at index Math.floor((n-1) / 2)
* How to add things to a Max Binary Heap?
  + Add it to the end of the array
  + Bubble up (swap value with ones already in the heap if necessary)
* Removing from a Max Binary Heap? (Extract Max)
  + Remove the root (max)
  + Replace with the most recently added
  + Adjust (sink down / bubble down)
* Priority Queues
  + A data structure where each element has a priority
  + Elements with higher priorities are served before elements with lower priorities
  + Can be implemented however you want, but heaps work well here
* Big O
  + Insertion = O(log n) for ALL CASES (b/c binary heaps are always balanced and as wide as possible)
  + Removal = O(log n) for ALL CASES (b/c binary heaps are always balanced and as wide as possible)
  + Search = O(n) b/c nothing is sorted with any logic

Hash Tables

* Hash tables are collections of key-value pairs
* Hash tables can find values quickly given a key
* Can add new key-value pairs quickly
* Hash tables store data in a large array, and work by hashing the keys
* A good hash should be fast, distribute keys uniformly, and be deterministic
* Separate chaining and linear probing are two strategies used to deal with two keys that hash to the same index
* How to deal with collisions?
  + Separate Chaining
    - At each index of the array hash table, we store values using a more sophisticated data structure (another array or a linked list)
    - This allows us to store multiple key-value pairs at the same index
  + Linear Probing
    - When we find a collision, we search through the array to find the next empty slot
    - Still can only have the specific number of elements = total length of array
* Big O of hash tables
  + Insertion = O(1) avg
  + Deletion = O(1) avg
  + Access = O(1) avg

Graphs

* Graphs are a collection of nodes and connections
* No starting / entry point like trees (even though a tree is a type of graph)
* Vertex = nodes
* Edge = connections
* Use for Graphs
  + Social networks
  + Location / mapping
  + Routing algos
  + Visual hierarchy
  + File system optimizations
  + Recommendation engines
* Types of graphs
  + Undirected - no direction associated with the edges (can go in either direction)
  + Directed - can only move in one direction from one node to another
  + Weighted - edges have associated values
* Storing Graphs
  + Adjacency matrix
  + 
  + Adjacency list
  + 
  + Could also use a hash table…
  + 
* Big O
  + 
  + Adjacency List
    - Can take up less space in sparse graphs
    - Faster to iterate over all edges
    - Can be Slower to look up specific edge
  + Adjacency Matrix
    - Takes up more space in sparse graphs
    - Slower to iterate over all edges
    - Faster to look up specific edge
  + Using Adjacency List for the class

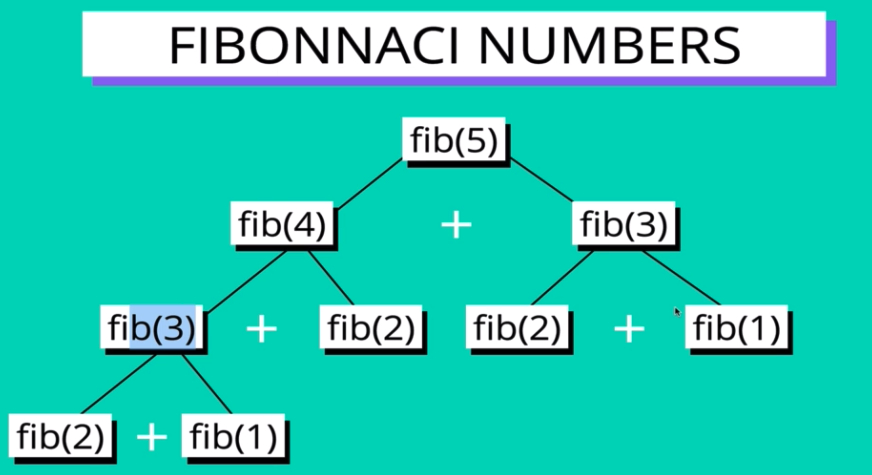
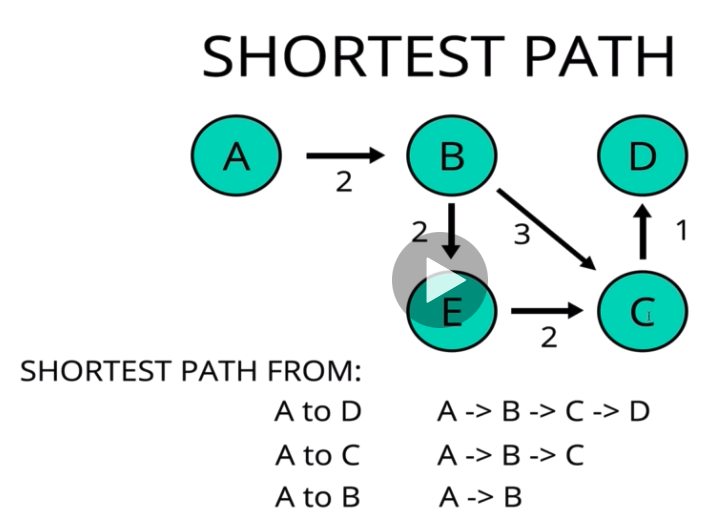
Graph Traversal

* Uses
  + Peer to peer networking
  + Web crawlers
  + Finding closest matches / recommendations
  + Shortest path problems (GPS, mazes, AI)
* Depth First Traversal
  + For graphs, we’ll explore as far as possible down one branch before “backtracking”
* Breadth First Traversal
  + Visit neighbor at current depth first

Dijkstra’s Algorithm

* Finds the shortest path between two vertices in a graph
* Need to add weight to our previous graph algo

Dynamic Programming

* A method for solving a complex problem by breaking it down into a collection of simpler subproblems, **solving each of those subproblems just once,** **and storing their solutions**
* Two uses:
  + Overlapping subproblems
    - Problem that can be broken down into subproblems which are reused several times
    - Examples: Fibonacci
    - 
    - See how fib(3), fib(2), etc. are repeated as things we need to do? This is where we can use dynamic programming!
  + Optimal substructure
    - Problem where an optimal solution can be constructed from optimal solutions of its subproblems
    - Example: shortest path from A to D in a graph
    - 
    - Can calculate the shortest path from A to D by calculating A-B, then A-B-C, then A-B-C-D!
      * Note, longest path doesn’t necessarily work like this!
  + Top Down Approach
    - The way you would normally solve Fibonacci using memoization
      * Start with what you‘re trying to find and working down to fill in the gaps and then adding everything together
      * Or, you could use…
  + Bottom Up Approach
    - Instead of memoization, use **tabulation**
      * Storing the result of a previous result in a “table” (usually an array)
      * Usually done through iteration
      * **Better space complexity** can be achieved through tabulation