Topics to cover:

Recursion

* find base case and return.
* otherwise, call the function again with certain parameters.
* if in a class: do you need to change actual information about the class? use reference parameters

Bio O:

deletion, insertion, search of a bst: O(logn)

Sorting

bubble or insertion sort: n^2 and stable

selection sort: unstable

* good sorting algorithims: nlogn

quicksort: nlogn average, unstable. merge is stable version and always but uses more memory

1. if array contains 0 or 1 element, return
2. select the first element
3. move all that are less to the left and all that are greater to the right
4. recursively repeat this process on the sub arrays

how to partition:

1. select first item as pivot
2. find the next item greater than pivot starting from left
3. find next item less than or equal to pivot on the right
4. swap the two items
5. continuously do while (low< high)
6. move pivot to right spot
7. return where pivot is

mergesort: split arrays in half and merge sort them when they come back you must merge

bubble sort: swap items going through the array and repeat until sorted

insertion sort: go to the next item and figure out where it goes in the already sorted items

selection sort: find smallest item and put in front

Trees:

* Traversal
  + preorder: process current, then left, then right
  + post order: process left, right, current
  + in order: process left, current, right
* Search, Insertion, Deletion
  + deletion:
    - find v and remember parent
    - case 1: node is leaf, just delete (unless root special case)
    - case 2: node has one child: just move child up
    - case 3: find right subtree smallest node, replace our node with that one's value. then delete that node

Tables, Hash Tables, Load

* hash tables - sorts values into buckets that can be accessed with an integer value
  + open – each value has a linked list after it with all values
  + closed – if value has duplicate hash, goes down the list until finding an open spot to store value

Priority Queues, Heaps, HeapSort

* maxheap keeps largest item at top
* complete binary tree: every row completely full, last row fills from left to right
* extraction (if empty or only one node special case)
  + remember top item
  + copy the value in the right most node on the bottom (last node)
  + delete that and put that value in the top
  + shift: swap the value with its larger children till it is the same size or bigger than children
  + return top value
* adding (if tree is empty special case)
  + insert in bottom left most empty spot
  + compare with its parent, and swap up if it is greater until in correct pos
* implementation: root is at array[0]
  + bottom node in array[count - 1], next empty spot is at count
  + left child = 2 \* parent + 1. right child = 2 \* parent + 2.
  + parent = (child - 1)/2
* heapsort: nlogn always
  + convert input into maxheap
    - starting at the n/2 - 1, focus on subtree rooted here
    - check that children are smaller, else shift down (same as extraction shift)
  + while numbers remain in the heap
    - remove biggest value and put it in the last spot of the array

Graphs:

* store in n by n array, leave point says if it connects to end point
* or use adjacency list, each vertex has a linked list saying if it connects to the others
* use array if they all connect or mostly, linked list if each only connects to few
* depth first traversal: if already visited, return. mark current as visited
  + process current vertex
  + for each edge, call depth first traversal
* using stack: push start on stack, then, while stack not empty
  + pop the top item, call it c
  + if c hasn't been visited, set as visited, and push each room leading from it onto the stack
  + use a queue for breadth first