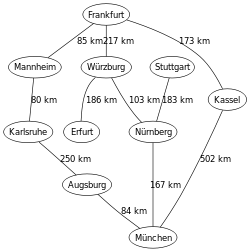
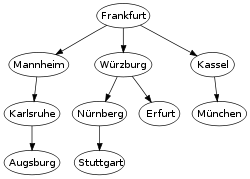
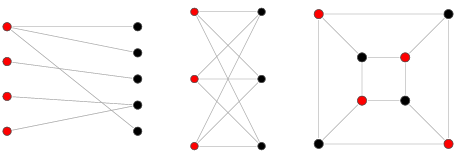
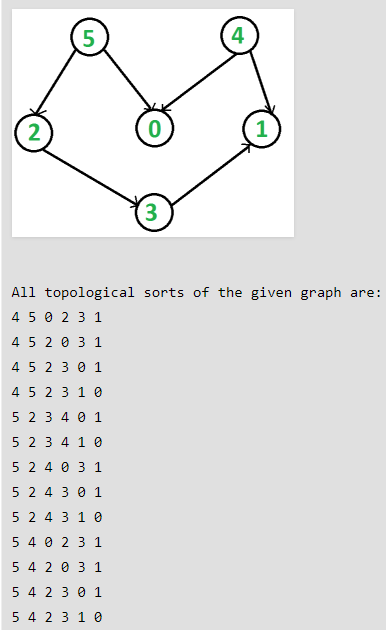
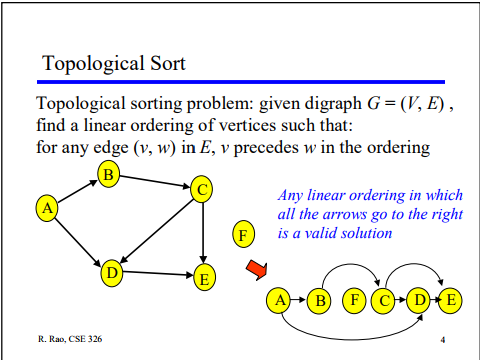
* Topological ordering
* Sorting: heapsort
  + Turn into heap (each node is higher value than its children)
  + Left child 2i+1
  + Right child 2i+2
* Shortest path
* MST
  + Prim’s
  + Kruskal’s
  + One more
* Connectivity
* Be able to analyze running time of algorithms (time complexity)

Make sure I can

* Given an algorithm, get its upper bounds, lower bounds, and tight bounds Big oh notation
  + Polynomial time
  + Logarithmic time
  + Exponential time
* Run stable matching on a set of men/women hospitals/patients
* Big-Θ is when the upper and lower bounds are the same
* Breadth first search, depth first search
  + Know how to do the topological ordering of each
  + Depth first. Start at bottom, move up
  + Breadth first – start at top of graph then move down a level starting at the left then moving right, mark the nodes
    1. 
    2. 
* Testing bipartite graphs
  + Meaning the graph can be broken up into two colors so that each point doesn’t point to one of its own colors
  + 
* Topological ordering
  + See where the graph can possibly start. Then check every possible combination
  + 
  + Look to see where one must start and must end; After that count the remaining nodes and do factoral of that number, check number of possible combos inside and do the previous factoral over the new
  + A sort in which all arrows go to the right is a valid solution
  + 
* Interval scheduling
  + Do greedily, first sort the schedules by whatever is prioritized, then go through the array and pick the compatible ones that come up first
  + Nlogn due to sorting, then n time to go through the array
* Interval partitioning – minimizing lateness
* Dijkstra’s algorithm
  + Starting node becomes 0 all others have infinity
  + Check all connected nodes with infinity
  + Have smallest route get set
  + When moving from node to node add the last one to determine the smallest
* Minimum Spanning Tree (MST)
  + Kruskal’s
    1. Sort all the edges in non-decreasing order of their weight.
    2. Pick the smallest edge. Check if it forms a cycle with the spanning tree

formed so far. If cycle is not formed, include this edge. Else, discard it.

* + 1. Repeat step#2 until there are (V-1) edges in the spanning tree
  + Reverse-Delete
    1. Start with graph G, which contains a list of edges E.
    2. Go through E in decreasing order of edge weights.
    3. For each edge, check if deleting the edge will further disconnect the graph.
    4. Perform any deletion that does not lead to additional disconnection.
  + Prim’s
    1. Create a set *mstSet* that keeps track of vertices already included in MST.
    2. Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.
    3. While mstSet doesn’t include all vertices
       - Pick a vertex *u* which is not there in *mstSet*and has minimum key value.
       - Include *u*to mstSet.
       - Update key value of all adjacent vertices of *u*. To update the key values, iterate through all adjacent vertices. For every adjacent vertex *v*, if weight of edge *u-v* is less than the previous key value of *v*, update the key value as weight of *u-v*
  + Clustering k-1 (after applying kruskal’s algorithm)
* Prefix codes (data compression) (Huffman)
* Huffman Encoding
  + 1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)
  + 2. Extract two nodes with the minimum frequency from the min heap.
  + 3. Create a new internal node with frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
  + 4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.
  + For characters and frequency order from smallest to largest
  + Get the 2 minimum and combine into a tree
  + Add those two minimum and their combined values goes back into the frequency order tree
  + Get the 2 minimum again where the above value is considered
  + Continue to get the 2 minimum until all are gone
  + Assign 0’s and 1’s to the edges to give each character its value
  + Uses priority queue
* Priority queue using a heap, array, linked list, sorted linked list and sorted array
  + Heap nlogn add/remove
    1. Optimum
    2. Add, put in last node, then sift it up
    3. Remove is the top node
    4. After top node is removed
       - Take last node, put it at top, then sift down
  + Linked lst, array adding 0(1) remove O(N)
  + Sorted linked list sorted array adding O(n) remove O(1)
  + Priority queue is least-first-out
    1. So, the smallest thing comes out first