# graphs

- 3 basic ways to represent a graph in memory (objects and pointers, matrix, and adjacency list)

1. objections & pointers

2. matrix

3. adjacency list

- Djikstra

1. initialize all vertex's distance to infinity

2. start somewhere, set distance to 0, set it visited

3. pick next unknown vertex v with shortest distance

- set that visited

- update its distance / path

- for each edge from v to adajenct unknown

- update v

4. repeat 3 until nothing not visited

- Prim's

1. start somewhere

2. find least outgoing edge of mst so far

3. add that vertex

4. repeat 2,3

O (E log E)

- Kruskal's

- grow a bunch of trees

O (E log V) - same asymptotic (log(V^2) = 2log(V))

- detect cycle

- dfs from every vertex and keep track of visited, if repeat then cycle

- topological sort - list vertices in order (all edges point in one direction)

- something with no edges can be put anywhere

- doesn't work iff there are cycles

- algorithm

1. calculate all indegrees

2. find node of indegree 0

- subtract from indegrees, all outgoing edges of 1st node

3. repeat 2 until no more nodes

Algorithms sheet

# recursion

- generate permutations - recursive, add char at each spot

- think hard about the base case before starting

- look for lengths that you know

- look for symmetry

- n-queens - one array of length n, go row by row

# searching/sorting

- binary search - use low<= val and high >=val so you get correct bounds

- **insertion sort** best when almost sorted

- **radix sort** best when small number of possible values

- **quicksort** usually fastest, but can be O(n^2)

- pick pivot, move things less than to left and things greater than to right

- returns void

- don't actually have to put pivot anywhere

- log n average extra space, sometimes n

- **merge vs. quicksort**

- On average, mergesort does fewer comparisons than quicksort, so it may be better when complicated comparison routines are used. Mergesort also takes advantage of pre-existing order

- quicksort is often faster for small arrays, and on arrays of a few distinct values, repeated many times

- **mergesort** can be parallelized, but usually uses extra space

- in place goes to n log^2(n)

- to implement, create a class so each method doesn't have to create its own array

- only extra memory is when merging (create temp array)

- **heapsort**

- n log n

- put all objects into a heap

- keep removing min and adding to array

- merge a and b sorted - start from the back

- binary sort can't do better than linear if there are duplicates

- if data is too large, we need to do external sort (sort parts of it and write them back to file)