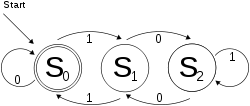
* Brute Force
* Divide and Conquer
* Decrease and Conquer
* Transform and Conquer
* Graphs
  + made out of vertices and edges
  + terms:
    - vertices
      * “points/places”
      * degree
        + number of edges incident upon it
        + indegree
        + outdegree
    - edges
      * “paths/roads/connections”
      * incident upon vertices
    - labelled
      * a graph where vertices have a unique name
    - weighted
      * a graph where edges have a value
    - directed
      * a graph where edges have a direction
      * arrows
    - simple
      * rules:
        + no reflexivity (irreflexive)
        + for undirected:

no two vertices have any more than one edge between them

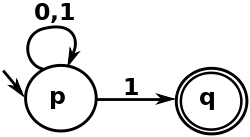
* + - * + for directed:

no two edges from the same source to the same destination

* + - cyclic
      * a graph with a path where nodes can be repeated
    - complete
      * a graph where there is an edge between all pairs of vertices
    - dense/sparse
      * dense
        + a graph where the number of edges is close to the maximum number of potential edges
      * sparse
        + a graph where the number of edges is not close to the maximum number of potential edges
    - planar
      * a graph that can be drawn without crossing edges
    - path
      * a series of consecutive edges
    - simple path
      * a path with no duplicates
    - cycle
      * a simple path with the only duplicate being that the source is the destination
    - tour
      * a cycle where every vertex is visited exactly once
    - length
      * it depends
      * could be:
        + the number of edges
        + the sum of the weights of the edges
    - disjoint
      * two paths/cycles that share no vertices/edges
    - connected
      * an undirected graph where there is a path between every pair of vertices
    - strongly connected
      * a directed graph where there is a path between every pair of vertices
    - subgraph
      * a collection of edges and vertices needed for the edges, from an original graph
    - connected component
      * a maximal connected subgraph
    - clique
      * a complete subgraph
    - spanning tree
      * the set of all vertices and enough edges to connect the graph as a tree
    - minimum cost spanning tree
      * the spanning tree with the least total weight/length
    - DAG
      * directed acyclic graph
    - network
      * a weighted digraph
  + maximum edges = V(V-1)/2
  + Representation
    - Adjacency List
      * an array linked lists
      * better for sparse graphs
    - Adjacency Matrix
      * a 2-D array with either a binary value for existing or not, or the value for the weight of the edge (with 0 or infinity for the edge not existing)
      * better for dense graphs
  + Traversals
    - Depth-first search (think road trips)
      * From a starting point v, (usually first alphabetical), go to next alphabetical neighbor of v, mark as visited.
      * Keep finding next neighbor of v alphabetically until you reach a dead-end. Backtrack to previous vertices and find another neighbor of v until all are visited.
    - Breadth-first search (think virus spreading)
      * From a starting point (usually first alphabetical), explore all next neighbors of v and queue them.
      * While the queue is not empty
        + dequeue vertex W and print it.
        + for next neighbor of W, N, if n is not visited, enqueue n.
    - \*Loop 1 and 2 to iterate through graphs
      * Loop 1:
        + for(int v=0; v<V; i++)
      * Loop 2: (v = current vertex you’re finding neighbors of)
        + for(int n=nextNeighbor(v,V); n<V; n=nextNeighbor(v,n))
    - nextNeighbor(uint v, uint w)
      * if(w == V)
        + w = 0;
      * for(int i = w + 1; i < V; i ++)
        + if adjacencyMatrix[v][i] is not visited or non existent, return i;
* Optimization Problems
  + Greedy Algorithms
    - takes the best possible choice at each step
  + Dynamic Programming
    - tries to solve the problem by solving subproblems
    - think of strong induction
    - Principle of Optimality
      * optimal solutions to subproblems can be combined to find the optimal solution for a larger problem
    - Steps
      * Does the Principle of Optimality apply?
        + if not, stop
      * Establish/find parameters to the problem
      * Write your Objective Function in terms of parameters
      * Solve Objective Function in terms of subproblems
      * Write n nested loops/recursions
* Graph Algorithms
  + Prims (greedy algorithm gives MST)
    - From starting point v, mark v as visited and check all neighbors of v.
    - Find smallest weighted edge and mark neighbor connected to the edge as visited
      * If edges have same weight, choose next alphabetical vertex
      * if smallest edge connects two vertices that are both marked as visited, ignore and chose next biggest edge
    - Look at all edges connected to the current tree, and repeat the process above of choosing the smallest edge.
    - return adjacency list/matrix of MST
  + Kruskal (greedy algorithm gives MST)
    - sort minheap of all edges
    - while the number of edges looked at is less than total edges in graph, get the next smallest edge and do a UNION-FIND to see if a cycle is not created.
    - if no cycle is created, add edge to MST until edges in MST = V-1
      * return adjacency list/Matrix of MST
  + Dijkstra (dynamic programming, finds shortest path from one node to all)
    - create a distances array, where index of starting point v has weight of 0 and the rest are infinity. Mark v as visited
    - update distance array with weights to all current neighbors of v. Find smallest distance from distance array and using its corresponding vertex, w, mark w as visited and update distances to all neighbors of w IF:
      * Distance to current node from v > distance from v to w + distance w + current node
  + Floyd (shortest path between two verticies)
    - array of shortest paths from a starting vertex to all vertices
  + Warshall (checks if graph is connected/trasitive)
    - returns true if the graph is transitive/connected, false otherwise
* Topological Sort
  + works on DAGs
  + determine indegree of each vertex
  + start at 0’s and add them to the queue
    - print first item in queue and adjust indegree of all nodes afterwards
  + done
* Max-Flow Networks
  + Use slow on edges such that flow is less than or equal to the weight of edges
  + Maximize Flow from source to destination
  + Sum of inflow equals sum of outflow
  + Ford-Fulkerson Algorithm
    - create residual network (graph with switched sources/destinations)
    - use findAugmentingPath
* P = NP
  + P = set of all deterministic TM that run in poly time steps
  + NP = set of all nondeterministic TM that run in poly time
  + P is a subset of NP
  + Turning Machines
    - Machine that uses an infinitely long strip of paper, can read, write, skip, go back to states on the paper. Can write 1’s and 0’s onto the paper.
      * Current state on reading head of TM determines what to do next
    - Every TM includes a finite-state machine (FSM)
      * Has limited number of states.
    - Deterministic finite-state automata(DFSA)
      * machine can go from state to state such that it doesn't have to decide which state to go to by itself. (No branching paths)
      * EX: this is a dfsa b/c given a string of 0s and 1s, it knows what state to go to.



* + - Non-Deterministic finite-state automata(NFSA)
      * machine cannot decide what is the next state to go to because of branching paths
      * EX: given a string of 0s and 1s, machine doesn’t know whether to loop or go to next state q.



* + Traveling Salesman
    - Given a list of cities and distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city.
      * NP-hard problem
    - Hamiltonian Circuit/Tour
      * graph cycle that visits each node exactly once.
  + Church-Turing theorem
    - If TM is capable of describing any algorithm procedure then there are problems with no algorithmic solution
  + Halting problem
    - No way of knowing whether any given program will eventually halt or run indefinitely. No matter how much time is passed, you cannot be sure if a program can be identified as non-halting or halting.
      * Cannot detect: endless number of cases that can make a program loop indefinitely, cannot check every single case.
    - Traceability
      * exponential time: untraceable /inefficient
      * poly time: traceable
  + NP-Complete
    - subset of TM in NP that can be converted into each other in poly time
    - Candidate Solution
      * potential solution to problem
    - Partial solution:
      * current work done towards candidate solution
    - Backtracking
      * “pruning” : getting rid of options that won't get to solution (like depth first search)
        + Think chessboard 4 queens problem
    - **Branch & Bound**
      * **used in optimization problems**
        + **objective function: see if solution is better than other**
    - **Minmax**
      * **best case then worst case alternate**
        + polynomial time
  + NP-Hard
    - every problem in NP can be reduced into probs at least as hard as NP
    - Reduction
      * inventing poly time algorithm that converts Problem A to Problem B. A cannot be harder than B, but B can be harder than A or equal in complexity.
  + Approximation vs Decision problems
    - Faster way to get to solution, not the best but close
    - Heuristic approach
      * way of making decisions but cannot prove it works (rule of thumb)
* Algorithm
  + series of well defined finite unambiguous steps that finds a correct solution
* B-Trees
  + a general tree where if a node has k trees, then it has k+1 children
  + traits:
    - perfectly balanced
    - searchable
    - restriction on number of keys (finite maximum size)
      * t - order of tree
      * keys: t - 1 <= current keys <= 2t - 1
      * (-------------------------)
        + does not apply for root
      * height is less than or equal to log base t of (n + 1)/2
    - keys are ordered
      * children are ordered relative to their parents
* B+Trees
* 2-3 Trees
  + b-trees where the number of keys for each node can only be 2 or 3
* 2-3-4 Trees
  + b-trees where the number of keys for each node can only be 2, 3, or 4
* Trees
  + Binary trees
    - Binary search trees
      * Binary trees where all items less than a node are on its left subtree and all items greater than or equal to a node are on its right subtree
    - Self-balancing binary search trees
      * AVL (Adelson-Velsky & Landis) trees
      * Red-black trees
  + Min heap
  + Max heap
  + Priority queues

*List of topics*

* Big-O notation, Big-Omega, Big-Theta
* Master theorem
* Trees
* Binary trees
* Binary search trees
* Binary trees where all items less than a node are on its left subtree and all items greater than or equal to a node are on its right subtree
* Self-balancing binary search trees
* AVL (Adelson-Velsky & Landis) trees
* Red-black trees
* Min heap
* Max heap
* Priority queues
* B-trees
* Sorting algorithms
* ~~Bubble Sort~~
* “While possible, when I put that in an exam, it's for negative points as a deterrent...” - Gabarró
* "Fuck bubble sort." - Gabarró
* Selection Sort
* Insertion Sort
* Shell Sort
* Radix Sort
* Merge sort
* Complexity is [Equation] (worst case)
* This is important because if a question on the test requires sorting, and it must run in *nlog(n)* time, merge sort or heapsort must be used.
* Quicksort
* Heapsort
* Complexity is [Equation] (worst case)
* Hashing
* Graphs
* Problem-solving techniques
* Greedy
* Dynamic programming
* Principle of Optimality
* Divide-and-conquer
* Example: merge sort
* Brute force
* Transform-and-conquer
* Example: Heap sort
* Use tools not designed to solve the problem by transforming it
* Backtracking
* Branch-and-bound
* Approximations
* Heuristics

*CS-182 Study questions & answers*

Master Theorem

* Write the Master Theorem equation.
* [Equation]
* What is the meaning of the first variable, [Equation]?
* The number of subproblems to look at.
* Another way to look at it is: how many recursive calls are made.
* What is the meaning of the second variable, [Equation]?
* Problem fractioning – how much smaller does your problem get each recursion?
* If the problem is halved each time, [Equation] would equal 2 – so [Equation].
* What is the meaning of the third variable, [Equation]?
* The complexity of non-recursive code.
* Example: two nested for-loops is probably[Equation], so [Equation].
* An example of a Master Theorem problem is getting the
* Write the three Master Theorem possibilities.
* [Equation]where [Equation].
* Just plug in the three variables to get your complexity.

Heaps

* What is always true for any given node in a min-heap?
* A node is always less than (or equal to) its children and greater than (or equal to) its parent.
* What is always true for any given node in a max-heap?
* A node is always greater than (or equal to) its children and less than (or equal to) its parent.

AVL trees

* What is the order of a binary tree?
* The height of the root’s right subtree minus the height of the left subtree.
* Should be -1, 0, or 1 in a balanced tree.
* What is done in a left-left problem?
* Rotate right on the problematic node (where the problematic node is the node which is not ordered)
* What is done in a right-right problem?
* Rotate left on the problematic node
* What is done in a left-right problem?
* Rotate left on the left child, then rotate right on the problematic node
* What is done in a right-left problem?
* Rotate right on the right child, then rotate left on the problematic node

Red-black trees

* What color does the root node have to be?
* Black.
* What color does a child start as when it is added?
* Red.
* What color must all leaves be?
* Black.
* What must always remain the same at all leaves?
* The number of black nodes between (and including) the root and the leaf.
* How is insertion of a node performed?
* Just insert the node like any other binary search tree, then color it red. Then check for problems.