Miscellaneous

- In dense graphs, O(E) = O(V^2)
- sum(i=0, n) r^i = r^(n+1) -1 / r - 1
- Remember b^logb(k) = k
- Examples of DP algorithms: Floyd-Warshall algorithm, LCS, Bellman-Ford algorithm
- Adjacency matrix: bad space O(V^2), can find neighbours in O(V), check edge in O(1)
- Adjacency list: Space O(V+E), O(V) or O(log V) to check if edge exists, O(X) to check all adjacent vertices
- For proving induction step: Need to represent as a value k (NOT N)
- Graph is sparse if E << V^2
- Time complexity of checking if edge exists in graph using adjacency list is O(log V) if you do binary search, O(V) without
- Graph representation is either adjacency list or matrix
- Fibonacci base case is fib(0) = 0, fib(1) = 1
- Recursive algorithms can pre-create an array before calling
- Can iterate through graph by "for V = 1 to N in G"
- Big-Theta is the lower bound
- Complexity definition: Think of the graph of the OG function being bound or binding another function
- Double check when checking the minimum value you initialise your value as Inf

Sorting Algorithms

- Recursive quicksort is not in place because space required for recursive call is not constant, is O(log N) because of the stack frames
- Counting sort with alphabets is O(N) because domain size is a constant
- For comparison-based sorting algorithms, still need to consider word size
- Merge sort space is O(n) and quick sort space is O(log n)
- Quicksort depends on two main factors: partition algorithm and pivot selection
- Quicksort is O(n log n) at worst when pivot is median, O(n^2) when pivot is smallest or largest in array
- Quicksort average case: After partitioning, can either be in green subarray (worst case partition size of 1 and N-1) or grey subarray (worst case partition size of N/4 and 3/4N). If pivot is always in green will keep partitioning into 3/4 N and stop at where size is 1 so (3/4)^h N = 1 -> N = (4/3)^h -> h = log4/3(N), max possible height when pivot in green is 2 log4/3(N) so O(log N)
- Counting sort and radix sort is done in linear time cos of the assumption that input is only integers in a small range
- Best-case input for sorting is usually when the list is already sorted
- Best case of heap sort is when every element is the same -> no need to down heap
- Heapify takes O(n) because nodes with a height of 1 need at worst O(1) operations and there are about N/4 of these nodes so O(N/4) + O(N/8) * 2 + ... + h * 1 which converges to N
- Can heapify an array in O(n) time so runtime for heapsort dominated by deletion operations
- Heap elements should start at 1, not 0
- Best case for insertion sort is O(n)

Recurrence Relations

- Proving recurrence relations by induction
    - Do base case
    - For inductive step, find the next step to go towards (typically k+1)
        - Can somethings be like k and 3k if T(n) = T(n/3) + a is given or something
    - Assume that T(k) = T'(k) is true → Need to prove that T(3k) = T'(3k) is true
        - Evaluate T(3k) using recurrence relation given
        - From T(3k) = T(k) + ~~, find an expression for RHS (T(k) + ~~ = T'(k) + ~~)
        - Try to solve T'(k) + ~~ using proposed solution
        - Try converting it to T'(3k)
- Remember to write constants a,b in recurrence relation
- Recurrence relation: Also need to consider that some operations may not occur in the base case that would normally occur in the recursive case
- Recurrence relations: When doing a single loop, need to specify a*N instead of just N because there is a constant operation being done
- Recurrence relations: Start with general expression and base case, find expressions for RHS terms and then sub into original equation, find pattern using k and try to get to base case, evaluate
- Recurrence relations: Need consider other operations that may be O(N) and so on -> RUN THE RAM MODEL
- For recurrence relations: When you have an expression like T(N/2) + aN, remember to substitute ALL N WITH N/2 WHEN FINDING T(N/2)

Binary Search

- Binary search invariant: key is in array[1...N] if and only if key in array[lo...hi-1]
- Binary search termination: lo = hi -1 because lo >= hi - 1 for while loop to terminate. Since lo < mid < hi -> lo < hi when while loop ends. So lo <= hi - 1 when while loops terminates. So based on two statements, loop ends when lo = hi – 1
- Invariant for binary search is key is in array[lo...hi)

Quickselect

- Quickselect: Call with k = N/2, Choose random pivot, partition it and if k == index(pivot), you found the median, if the k > index(pivot) then need to search RIGHT sublist, or else need to search left sublist
- Quick select, remember to specify N//2 to find the median of the list
- Average case of O(n) as
- $T(n) = n + 0.75n + 0.75^2 n + 0.75^3 n + \dots$
- $T(n) = (1 + 0.75 + 0.75^2 + 0.75^3 + \dots)n \leq \left(\frac{1}{1-0.75}\right)n = 4n.$

Prefix Trie/Suffix Trie

- Longest repeated substring in suffix trie: Find deepest node with at least 2 children

Suffix Tree

- Suffix tree: Longest repeated substring involves finding the deepest (so you can find the longest) node with at least 2 children
- Suffix tree is a compact suffix trie (compressed to (index, length) as the node)

- Suffix tree is O(n) space as the total number of leaf nodes is O(n) and each node has at least 2 children so complexity is O(n + n/2 + n/4) == O(n)

Suffix arrays

- To make suffix array -> get all suffixes (including $). create rank array -> sort on 1st,2nd,4th,...,N/2,N characters, if current ranks are same, suffix with smaller rank is smaller. if current ranks are the same, first k characters must be the same
- Suffix array substring search: do binary search
- Suffix array construction is now O(N log^2 N) and will take O(N) space
- Prefix doubling for suffix arrays reduce construction cost to O(N log^2 N) (each sorting takes O(N log N) and you sort 1,2,..,N/2, N characters)
- Suffix array: Must include "$" position in the array
- Space complexity of building suffix array needs original string, rank array and suffix array which are all O(N) so O(N) in total

Dynamic Programming

- Edit distance: situations are letters matching, substituting s1[n] into s2[m], adding s2[m] into s1[n], removing s1[n] (ALL THE COMPROMISES ARE DONE TO FIRST STRING)
- Edit distance base case: same as length of word (memo[i,0] = i; memo[0,j] = j)
- Longest common subsequence: Recurrence relation is if letters match then its LCS(i-1, j-1) if they dont its max(LCS(i-1, j), LCS(i, j-1))
- Unbounded knapsack is the single axes problem, 0/1 knapsack is the double axes problem
- Top-down DP is basically bottom-up DP but when assigning to a past value, you instead call the DP method to get that value
- When discussing why recursive implementation for DP is bad -> Mention repeating calculations and provide example

AVL trees

- Right-right case for AVL is when node has BF of -2 and right child has BF of 0 or less
- AVL trees: Whoever becomes the head honcho says 'take this L' to the previous head honcho

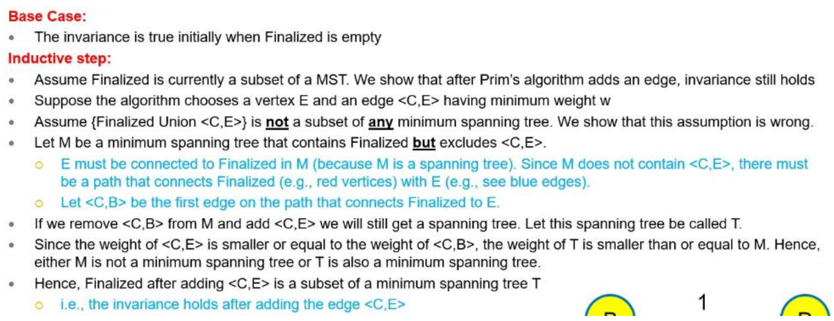
DFS/BFS

- BFS involves adding src to discovered, for every adjacent edge, add undiscovered vertices to discovered, then put the vertex in finalised, keep going
- BFS and DFS complexity is O(V+E), visits every vertex once and every edge at max twice
- DFS, start at src, mark vertex as visited, for each edge call DFS(vertex)
- Can use DFS to find cycles if it sees an edge that leads to an already visited node, making sure to ignore the vertex it came from
- DFS and BFS are both O(V+E) if using adjacency list

Prims/Kruskals

- "Union-Find has two operations SET_ID(u) which finds which set a vertex belongs to (O(1)) and UNION_SETS(Si, Sj) which combines two sets in (O(V log V))
- Both Prims and Kruskals uses the same proof by contradiction in inductive step
- Prims is very similar to Dijkstra's except when you insert/update an entry, it's the weight of the edge NOT the weight + u.distance

- Prim's also adds the edge to discovered, NOT just the vertex. When adding to finalized, just add the edge (WITHOUT THE WEIGHT)
- Prim's Invariant: Finalized is a growing subset of a minimum spanning tree
- Kruskal's algorithm: Finalized is a growing subset of a MST
- Kruskal's is O(V log V) for union_sets as height of O(log V) with O(V) to combine two lists
- Prims: Need to show that after you add an edge to the graph, it's still an MST
- Kruskals: Need to show that between two disjoint sets, after you add an edge to join the two, it is still an MST
- Need to include base case when showing invariant proves the algorithm
- Prim's



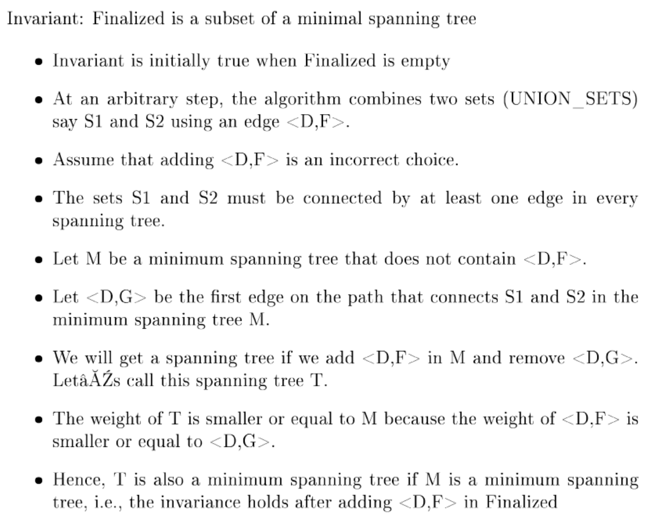
Base Case:

- The invariance is true initially when Finalized is empty

Inductive step:

- Assume Finalized is currently a subset of a MST. We show that after Prim's algorithm adds an edge, invariance still holds
- Suppose the algorithm chooses a vertex E and an edge <C,E> having minimum weight w
- Assume {Finalized Union <C,E>} is **not** a subset of **any** minimum spanning tree. We show that this assumption is wrong.
- Let M be a minimum spanning tree that contains Finalized **but** excludes <C,E>.
    - E must be connected to Finalized in M (because M is a spanning tree). Since M does not contain <C,E>, there must be a path that connects Finalized (e.g., red vertices) with E (e.g., see blue edges).
    - Let <C,B> be the first edge on the path that connects Finalized to E.
- If we remove <C,B> from M and add <C,E> we will still get a spanning tree. Let this spanning tree be called T.
- Since the weight of <C,E> is smaller or equal to the weight of <C,B>, the weight of T is smaller than or equal to M. Hence, either M is not a minimum spanning tree or T is also a minimum spanning tree.
- Hence, Finalized after adding <C,E> is a subset of a minimum spanning tree T
    - i.e., the invariance holds after adding the edge <C,E>

- Kruskal's



Invariant: Finalized is a subset of a minimal spanning tree

- Invariant is initially true when Finalized is empty
- At an arbitrary step, the algorithm combines two sets (UNION_SETS) say S1 and S2 using an edge <D,F>.
- Assume that adding <D,F> is an incorrect choice.
- The sets S1 and S2 must be connected by at least one edge in every spanning tree.
- Let M be a minimum spanning tree that does not contain <D,F>.
- Let <D,G> be the first edge on the path that connects S1 and S2 in the minimum spanning tree M.
- We will get a spanning tree if we add <D,F> in M and remove <D,G>. LetâĂŹs call this spanning tree T.
- The weight of T is smaller or equal to M because the weight of <D,F> is smaller or equal to <D,G>.
- Hence, T is also a minimum spanning tree if M is a minimum spanning tree, i.e., the invariance holds after adding <D,F> in Finalized

BWT

- Naive method for BWT inversion: Sort BWT -> Do BWT + result -> Sort it -> Do BWT + result -> ... -> Get first row
- Last-First Property: The last character in the row comes before first column
- Can decompress BWT because relative order of letters remains the same
- BWT: Efficient method number each char in last column by relative order, row = 1 (want to start with character before $ which is always in first row), str = $, iterate until word is made -> {get letter based on row and add to str, row = rank[c] + num(c) – 1 (basically offsetting downwards from the first occurrence of that letter)
- BWT Substring finding: Go backwards from string, find first and last occurrence of letter, relate that to the first column (that is your range), repeat until string is finished or no match is found
- BWT: Number of occurrences of first letter after finding substring in BWT is the number of occurrences

- Efficient method: Relies on fact that relative orders of letters in first column is same with the last column

Hashing

- Cuckoo hashing: Two tables with two hash functions, when inserting it kicks out w/e was there and that item has to find a new home, keeps going
- Double hashing found by: Index = (hash1(key) + i*hash2(key)) % M

Single-Source All Targets

- BFS (shortest path): Start with src, for each outgoing edge, if not discovered/finalised u.distance = v.distance + 1, add to discovered
- Can do single source all targets problem with BFS, Dijkstra's algorithm, Bellman-Ford algorithm
- Dijkstras: Visit each edge once O(E) and while loop runs for O(V) times, searching for vertex with smallest distance O(V) so total O(E+V^2) = O(V^2)
- Dijkstra's algorithm consists of finding smallest in discovered AND updating a given entry's distance
- Prove disjkstra's with proof by contradiction
- Dijkstra's: Make sure you insert the entry in discovered as v.distance = w
- Dijkstra's alg does not work with graphs with negative weights (not cycles)
- Dijkstra's algorithm is a greedy algorithm
- Dijkstra's preferred if no negative edges, or else use Bellman-Ford
- Bellman-Ford basically relaxes all edges continually for V-1 times and then checks for any possible relaxations after that (if it does occur there is a negative cycle)
- Bellman-Ford algorithm: Need to include and update pred
- Space complexity of BF algorithm is O(V+E)
- Bellman-Ford algorithm solves single source, all targets problem for graphs with negative weights (returns error if it has negative cycles)
- Bellman-Ford, need to draw table from 0...V-1 for each vertex WITH one more iteration to check for negative cycles

All-Pairs Shortest Paths/Transitive Closure

- Think of Floyd-Warshall's algorithm as the middleman-algorithm
- For every middleman vertex k, for every pair of vertices update the distance of as the min(current distance OR going through middleman)
- If floyd-warshall algorithm has a negative cycle will have vertex v such that dist[v][v] is negative
- Floyd-Warshall and Transitive Closure both have complexities of O(V^3) time and O(V^2) space
- Transitive closure is just a modification to floyd-warshall by pre-processing True between vertices that are already connected or vertices that are the same
- Only time you use adjacency matrix for graphs is during Floyd-Warshall algorithm or transitive closure
- Floyd-Warshall is an all-pairs shortest path algorithm, use Dijkstra's or BF for single source
- Want to use adjacency matrix for transitive closure if runtime is important and graph is not sparse

- Transitive closure first checks True for every adjacent vertex and for the vertex itself, then checks for every vertex, and every pair of vertex to see if it can get to the other via the original vertex

Network-Flow

- Flow network has 2 properties: flow is always less than capacity, flow in a vertex = flow out of a vertex
- Be careful when augmenting flow → If taking a path that goes up a negative route, need to minus flow
- Flow of network is total flow out of source vertex
- Ford-Fulkerson Algorithm works by 1) create residual network 2) find augmenting path 3) augment graph based on residual capacity 4) update residual network
- Capacity of a cut is the total capacity of outgoing edges
- Flow of cut is total outgoing - total incoming
- Flow of the network = Flow of any cut
- Max flow of a network is equal to min-cut and can be found using FF algorithm
- Ford Fulkerson Complexity is O(EF) where F is the max flow of the network. Each iteration of while loop is O(V+E) and this repeats O(F) times -> O(F(V+E)) -> O(EF) as O(E) >= O(V^2) for connected graphs
- When FF terminates, there is a cut where both conditions are true
- Finding min-cut: Solve FF to find residual network with no augmenting paths then find vertices reachable from s (this forms cut-set S)

Top Sort

- When drawing DAG according to criteria, need to include every point
- Can always just use DFS for top sort