CS2020 Quiz 2 Cheat-sheet

1. Graph Theory

Search in a graph	1. Depth-first search (DFS): Traversal by path, similar to pre/in/post-order traversal for trees, use a *stack* to have iterative implementation, easy to have recursive implementation, have pre / post-order versions, visit each vertex and node exactly once, produce a DFS tree (or forest if the graph is not connected), time complexity: O (V + E); 2. Breadth-first search (DFS): Traversal by level, use a *queue* to have iterative implementation, hard to have recursive implementation, visit each vertex and node exactly once, produce a BFS tree (or forest), time complexity: O (V + E).
Detect cycles in a graph	1. In an undirected graph: use simple *DFS* or *BFS*, a cycle is found when a visited node is explored again, time complexity: O (V + E); 2. In a directed graph: 1) Use *classified DFS* to find tree edges and back edges (timestamp the discover time and finish time for each node). Discovery of cycles is equivalent to discovery of back edges. An Edge (u, v) is a back edge if and only if $d[v] < d[u] < f[u] < f[v]$. Time complexity: O (V + E); 2) Use *topological sort*, very similar to *classified DFS*, simplify the timestamp process, time complexity: O (V + E); 3) Use *Tarjan's algorithm*, time complexity: O (V + E).
Topological sort (in DAG)	1. Post-order DFS: Run post-order DFS on a DAG and output the vertices in reverse order (use a stack to implement this) of finishing time, time complexity: O (V + E); 2. Khan's algorithm (BFS): Record the in-degrees of all nodes in an array, then enqueue all nodes with in-degrees of 0. Dequeue one node with in-degree of 0, and update the in-degrees of all its neighbors. If any neighboring node has an in-degree of 0 now, enqueue it. Time complexity: O (V + E).
Shortest path in a graph	1. Bellman-Ford algorithm: Relax all edges for k times (k is the number of edges in the graph). Terminate earlier if a whole round does not change any estimate distance. Work for negative weights. Time complexity: O (EV); 2. Dijkstra algorithm: For graphs without negative weights, each time remove the node with shortest distance in the priority queue, relax and add its all adjacent nodes to the priority queue, optimized time complexity: O(E * logV); 3. Constant-weight graph: If all edges have the same weight, use simple BFS to find the shortest path (because the minimum number of hops is equivalent to minimum distance now); 4. Directed acyclic graph: In a DAG, shortest path can be found via topological sort. Get the topological order of the graph and relax in that order. Relax all edges of each node iteratively in topological order. Time complexity: O (V + E); 5. Undirected tree: In a (undirected, acyclic) tree, we can use simple BFS or DFS to find shortest path, time complexity: O (V); 6. Negative cycle: Run *Bellman-Ford* for k+1 rounds, a negative cycle is detected if there are still changes in estimated distances; 7. Longest path in a DAG: negative all weights and use the same topological sort method; 8. Single source to all destinations: Standard algorithms can be used, just avoid early termination for Dijkstra's algorithm; 9. Multiple sources to single destination: Reverse the sources and destination and use #8; 8. Multiple sources to all destinations: Use the super-source method.

Minimum spanning tree (MST)	1. <u>MST properties</u>: 1) No cycles; 2) If a MST is cut, we can get 2 MSTs; 3) For every cycle in the graph, the maximum weight is not in MST; 4) For every component in the graph, the minimum weight across the cut is in MST; 2. <u>Prim's algorithm</u>: Similar to Dijkstra, each time add the minimum weight across the cut (the edge and the node on the other side). In the meantime, use a priority queue to store the distance to the existing part of MST for each node. Optimized time complexity: O(E * logV); 3. <u>Kruskal's Algorithm</u>: Sort all edges according to their weights. In such an order, add each edge to the MST if they do not form a cycle. Use union-find, if two nodes of an edge is in the same set, then we will not add because this will form a cycle. Time complexity: O (E * logV); 4. <u>Borůvka's algorithm</u>: Start from each node itself as a component. In each round, add the minimum weight outgoing edge for each component to merge each other so that only half of the components left. Time complexity: O (E * logV); 5. <u>Constant-weight graph</u>: Use simple *DFS or BFS*. The resulting DFS or BFS tree is just MST; 6. <u>Directed acyclic graph with a root</u>: Add minimum weight incoming edge for every node except the root, time complexity: O (V + E); 7. <u>Maximum spanning tree</u>: Negate all edges and run normal algorithms.
Graph modelling	1. Meet in the middle: Determine the shortest path when a certain edge has to be passed by; 2. Super source: Multiple source to all destinations; 3. Duplicate: When some nodes have special cases; 4. Vertex contraction: Merge two edges into one.

<u>2. Tree, Heap & Union-find</u>

Binary Search Tree (BST)	1. <u>Terminology</u>: root, leaf, internal, parent, child, ancestor, descendant, predecessor, successor, ancestor, descendant, subtree; 2. O (h) operations (become O (logn) for balanced trees): search, insert, delete, predecessor, successor; 3. O (n) operations: pre/in/post-order traversal.
AVL Tree	Need 1 or 2 times of *rotation*s to keep balance.
Augmented Tree	1. <u>Rank tree</u>: Store size of the sub-tree rooted at each node; 2. <u>Interval Tree</u>: Store interval at each node and use the starting point as the key. Store the maximum end point of the sub-tree at each root for search.
Heap	1. <u>Max/min heap</u>: parent is larger/smaller than both children, no relation guaranteed between children; 2. Need O (n) time to find *predecessor/successor*; 3. <u>Heap sort</u>: divide into two process, *heapify* needs O (n) time, while *extraction* needs O(n logn) time; 4. Heap can be used to implement *priority queue*, *heap-of-heaps* is useful for some problems.
Disjoint Set	1. Quick find: find - O(1), union - O(n); 2. Quick union: find - O(n), union - O(n); 3. Weighted union: find - O(logn), union - O(logn).

<u>3. Hashing</u>

Hashing Theory	1. Hash function: use hashcode() method to get the x for hash function, and then match y of hash function to the slot index; 2. Collision: chaining (use a linked list at each slot, Java adapted), open addressing (linear probing, double hashing); 3. Simple uniform hashing assumption: every key is equally likely to map to every bucket independently; 4. Table resizing: If n == m, then double the table; if n < m/4, then halve the table.
Fingerprint and Bloom filter	1. Fingerprint has false positive but no false negative; 2. Bloom filter uses multiple hash functions to decrease the probability of false positives.

---- End ----