CS2020 Final Cheat-sheet

1. Java Language Specification

Class & Interface	1. Implementation: A non-abstract class should include all methods of the interface being implemented, and the *return type* and *signature* for all methods must be identical; 2. Class: A class can implement multiply interfaces and inherit only 1 superclass. Inner class can be used to implement multiply inheritance. 3. Interface: All fields in an interface are *public static final*, no matter declared explicitly or implicitly. All methods in an interface should be *abstract* and *non-static*. 4. Constructor: No constructor should appear in an interface. A class can have multiply constructors, all of which must have different signatures. If superclass does not explicitly define its default constructor, all constructors from its subclasses must explicitly call one of the super constructors as the first statement.
Inheritance	1. An interface can inherit multiply interfaces, but a subclass can only inherit one super-class. 2. Subclass substitution: The *running-time type* of a variable can be the subclass of its *compiler-time type*. 3. Polymorphism: Because of subclass substitution, we are not able to check which version of the method is called at compiler-time. Thus, we can only call (*without explicit type cast*) the methods of the *compiler-time type*. 4. Override: Method override cannot change its return type and signature, and can only throw the same or subtype of the original exception (unless runtime exception). JVM always tries to call the overriding method in the subclass whenever possible. Final methods can only be overloaded rather than overridden. Static methods are always seen as new methods. 5. Overload: Method overload has to change its signatures (number, type or order of parameters). Return type can be changed optionally but cannot be used to differentiate.
Access modifier	1. Static: Belongs to the class itself rather than any object instantiated by the class. *Constructor* can *never* be *static*. *Local* variables cannot be static. Static method cannot call non-static fields, while non-static methods are free to call static fields. Static method can only be called directly or use class name, cannot be called using `super` keyword. *Non-static inner class* cannot exist static fields or methods. 2. Final: Final variables cannot be re-assigned value, but if it is referring to an object, things inside that object can be changed. Final methods can only be overloaded rather than overridden. All fields in an interface are final implicitly. 3. Private: Private fields or methods can only be called inside its class (and its outer class if its own class is an inner class). Methods in an interface cannot be private. 4. Protected: Can be called by itself and its subclasses. A method overriding a protected method cannot be private.
Exception	In Java, exception consists of *IO exception* and *runtime exception*. It is compulsory to check IO exception *only*. As long as there is potential *IO exception*, all methods have to surround it with *try/catch* or declare it.

2. Time Complexity Analysis

Recurrence Tree	1. Method: Every recurrence relationship can be written in the form of a tree. What we need to is: 1) expand the tree big enough; 2) find the sum of *each level*; 3) find *how many levels* there are. In this way, the only last thing is to calculate the sum of first n terms for a certain mathematical series. 2. Mathematical series: 1) $1+2+\cdots+n=O(n^2)$; 2) $a+a\cdot q+\cdots+a\cdot q^n=a\cdot\frac{1-q^n}{1-q}$, it becomes $\frac{a}{1-q}$ when $\lvert q\rvert<1$; 3) *Harmonic series* $1+\frac{1}{2}+\cdots+\frac{1}{n}=\log n$.
Master Theorem	If the recurrence relationship is in the form of $T(n)=a\cdot T\left(\frac{n}{b}\right)+f(n)$, let $x=n^{\log_b a}$ and compare it with $f(n)$: 1. If $x>f(n)$, then $\mathrm{T(n)}=\Theta(n^{\log_b a})$; 2. If $x=f(n)$, then $\mathrm{T(n)}=\Theta(f(n)\cdot\log n)$; 3. If $x<f(n), a\cdot f\left(\frac{n}{b}\right)<f(n)$, then $\mathrm{T(n)}=\Theta(f(n))$;

	<u>Notice:</u> The relationship of > = < here is comparing the order of functions. For example, although $x^3 > x^2$ seems normal, $n \equiv n \cdot \log n$ as logarithmic part is neglectable.
Bound	Upper – $O(n)$ Tight – $\Theta(n)$ Lower – $\Omega(n)$

3. Linear Data Structure & Algorithms

Array, linked list, queue, stack	1. <u>Sorted array</u>: need $O(n)$ to insert, $O(\log n)$ to search; 2. <u>Unsorted array</u>: need $O(1)$ to insert, $O(n)$ to search; 3. <u>Linked list</u>: need $O(1)$ to insert, $O(n)$ to search; 4. <u>Queue</u>: have enqueue and dequeue, used in BFS; 5. <u>Stack</u>: have push and pop, used in DFS.
Search	1. <u>Linear search</u>: traverse the whole list from one end to the other, useful in greedy algorithms, need $O(n)$ time. 2. <u>Binary search</u>: basic idea of divide-and-conquer, list has to be sorted, need $O(\log n)$ time. 3. <u>Quick select</u>: randomly select pivots to separate the unsorted list, find out the k^{th} item or the first k^{th} largest / smallest items, need amortized $O(n)$ time. 4. <u>Peak finding</u>: 1) For 1D array, binary search and recurse on which half the larger neighbor is in, need time $O(\log n)$; 2) For 2D array, start to find the global maximum in the mid column and its two neighbors, recurse on which half the larger neighbor is in, need time $O(n \log m)$; 3) For 2D array, start to find the global maximum grid in the middle column, compare the left/right neighbors of this grid, recurse on the larger side, need time $O(n \log m)$; 4) For 2D array, divide intro 4 parts, find the global maximum on the border and cross, recurse on the part who is larger than that maximum, need time $O(m+n)$. 5. <u>Herbert log</u>: Use binary search but recurse on both half. Skip the whole segment if the start and end are the same. 6. <u>Aggressive cow</u>: Instead of dividing into two halves, keep multiplying by 2 until find the correct range and binary search on this range.
Sorting	0. The lower bound of *compare-based sorting* is $\Omega(n \log n)$. 1. <u>Selection sort</u>: In each round, linear search all the items to find the suitable one and swap to its position. The *first* few items will be sorted, the rest items remain almost unchanged while a few of them have exchanged their positions *in pair*. Best / worst time are both $O(n^2)$, unstable, in-place. 2. <u>Bubble sort</u>: The *last* few elements have been sorted, the smallest few elements have been shifted forward. They key idea is that all elements can only *bubble instead of jump*. The maximum possible times of shifting forward equals the number of element that have been sorted. Best time is $O(n)$, worst time is $O(n^2)$, stable, in-place. 3. <u>Insertion sort</u>: The *first* few elements have been sorted, the rest are *unchanged*. Notice that the first element may not be the smallest among the array, it may only be smallest in the sorted part of the array. Best time is $O(n)$, worst time is $O(n^2)$, stable, in-place. 4. <u>Merge sort</u>: Use divide-and-conquer, always divide into two halves and merge them. Best / worst time are both $O(n \log n)$, stable, not in-place (at least $O(n)$ extra space), used in counting inversions. 5. <u>Quick sort</u>: Partition around the pivot and recurse on both halves, use two pointers to swap to partition in-place, use pack-duplicate or maintain-four-region to handle duplicates, use random pivot to avoid the worst case, use paranoid quick sort to ensure the order of time complexity, dual pivots to further improve. Average time is $O(n \log n)$, unstable when there are duplicates, in-place. 6. <u>Heap sort</u>: have two parts, first part is heapify, similar to merger sort, keep joining two smaller heaps into one, need time $O(n)$; second part is sorting, to extract the maximum one by one, need time $O(n \log n)$. Total time $O(n \log n)$, unstable, in-place. 7. <u>Reversal sort</u>: use divide-and-conquer or quick sort. 8. <u>Random shuffle</u>: Use Knuth's algorithm with time $O(n)$. 9. <u>Convex hull</u>: 1) brute force – check for each pair (u, v), whether there is another point w such that $(\mathrm{u}, \mathrm{w}, \mathrm{v})$ is clockwise. If such w does not exist, then it's on the hull, need time $O(n^3)$; 2) selection sort –find the next point on the hull one by one, need time $O(nh)$, h is the number of points on the hull; 3) merge sort – choose a vertical line to divide these points into two halves, recurse on both and connect two parts, need time $O(n \log n)$; 4) quick sort – keep building triangles and delete interior points, need average time $O(n \log n)$.

Skip List	Build $\log n$ levels of linked lists, always go from high-level to low-level to search, randomize to insert into higher-levels, all operations need time $O(\log n)$.

4. Tree & Heap

Binary Search Tree (BST)	1. Terminology: root, leaf, internal, parent, child, ancestor, descendant, predecessor, successor, ancestor, descendant, subtree; 2. $O(h)$ operations (becomes $O(\log n)$ when balanced): search, insert, delete, predecessor, successor; 3. $O(n)$ operations: pre/in/post/level-order traversal
AVL Tree	1. When insert / update, needs 1 or 2 rotations to balance. When delete, may need up to $O(\log n)$ rotations; 2. Can store *pointers* in the nodes to make predecessor and successor query becomes $O(1)$; 3. Can use *adjacent hash table* to make search $O(1)$.
Augmented Tree	1. Rank tree: Store size of the sub-tree rooted at each node; 2. Interval tree: 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 (if search goes to left, safe to go to left); 3. Range tree: Store the maximum of the left subtree at each node, keep finding the split point to examine the range.
Binary Heap	Heap is usually stored in an *array* (by *level order*), support insert, delete and update in $O(n \log n)$, does not support search, predecessor, successor and traverse. Can use an adjacent hash table to support search. Can be used to implement *priority queue* and *heap sort*, can be further improved using *Fibonacci heap* and *heap-of-heaps*.

5. Hashing

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 (used by open addressing).
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.
Cuckoo Hashing	Use two separate tables with two independent hash functions. Push the original item to the other table if collision happens.

6. Graph Theory

Graph Search	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 implement recursively, 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: $O(V+E)$, space complexity: $O(E)$; 2. Breadth-first search (DFS): Traversal by level, use a queue to have iterative implementation, hard to implement recursively, visit each vertex / node once, produce a BFS tree (or forest), time complexity: $O(V+E)$, space complexity: $O(V)$.
Connected Component	Use union-find data structure (or known as disjoint sets), *quick find*: find - $O(1)$, union - $O(n)$; *quick union*: find - $O(n)$, union - $O(n)$; *weighted union*: find - $O(\log n)$, union - $O(\log n)$, can be further improved with *path compression*.
Detect Cycles	1. In an undirected graph: use simple *DFS* or *BFS*, detect a cycle 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	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: Simply use BFS or DFS to find shortest path in an (*undirected, acyclic*) tree, time complexity: O (V); 6. Negative cycle: Run *Bellman-Ford* for k+1 rounds, a negative cycle is detected if there are still changes; 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; 10. Multiple sources to all destinations: Create a super-source; 11. Shortest path within k steps: Duplicate the graph for k times, all edges point from level n to level n+1 (becomes a DAG, use topological sort). Time complexity: O(kV + kE); 12. All pairs shortest path: Floyd-Warshall algorithm, add the number of hops gradually, store in a 2D array, need time $O(V^3)$; 13. All pairs shortest path within k steps: Run #12 for all different steps [1…k], store in a 3D array, query only need time O(1), pre-process needs time $O(n^4)$.

Minimum Spanning Tree (MST)	1. MST properties: 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. Prim's algorithm: 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. Kruskal's Algorithm: 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. Borůvka's algorithm: 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. Constant-weight graph: Use simple DFS or BFS. The resulting DFS or BFS tree is just MST; 6. DAG with a root: Add minimum-weight incoming edge for every node except the root, time complexity: O (V + E); 7. Maximum spanning tree: Negate all edges and run normal algorithms.
Network Flow	1. Ford-Fulkerson algorithm: Add backward edges for each existing edge. Keep finding augmenting path (via DFS or BFS), compute its bottleneck capacity (ordered traversal), subtract that value for all forward edges on that path and add that value for all backward edges on that path. Only terminates for integer flow (or well-approximate for real numbers). Time complexity: O(F*E), where F is the maximum flow on an edge. 2. Max-flow / min-cut theorem: Flow f is a maximum flow if and only if there are no more augmenting paths in the residual graph. The value of that maximum flow equals the minimum capacity of an st-cut on that graph. 3. Electrical distribution problem: create a super-node for all power plants and another super-node for all consumers.