Disjoint Sets and Asymptotics

- We want algorithm to scale better in the worst case, e.g. Parabolas N^2 grow faster than lines N
- (Painful) Analysis − Simplification Summary
  - Only consider the worst case
  - Pick representative operation(s) (aka: cost model)
  - Ignore lower order terms and/or multiplicative constants
- Big-O and Big-Θ
  - Big-O : $\sim \leq$ (upper bound); $\sim$ worst case, $\neq$ worst case
  - Big-Θ : $\sim$ Equality
  - $\lfloor f(N) \rfloor = \Theta(f(N)) \lceil f(N) \rceil = \Theta(f(N))$
  - $\log_p(N) = \Theta(\log_q(N))$
  - Merge Sort: $\Theta(n \log n)$; Selection Sort: $\Theta(n^2)$
  - N^2 to $N \log N$ is enormous, but $N \log N$ to N is not much different.
- Master Theorem : Let $T(N) = a\,T(N/b) + \Theta(N^d)$, then

$$\begin{aligned} T(n) &= \Theta(N^d) && \text{if } \ d > \log_b a \\ &= \Theta(N^d \log N) && \text{if } \ d = \log_b a \\ &= \Theta(N^{\log_b a}) && \text{if } \ d < \log_b a \end{aligned}$$

- For constant c, then $O(1) \subset O(\log n) \subset O(\log^c N) \subset O(n) \subset O(n^c) \subset O(c^n) \subset O(n!) \subset O(n^n)$
- Disjoint Sets design + Runtime for connect() and isConnected():

  Quick Find $(\Theta(N), \Theta(1))$ → Quick Union ($\Theta(N)$ both since balance is not guaranteed; worse than QF) → Weighted Quick Union (WQU) ($\Theta(\log N)$ both, with $\Theta(N)$ creation) → WQU with Path Compression (iterated log).
- WQU must have height $h \leq \log_2(N)$; Path Compression only happens when doing $find()$
- QU, WQU etc. if connect(a, b), then b is above a; for WQU, in the array representation, the root node has value of size negated
- An Abstract Data Type (ADT) is defined only by its operations, not by its implementation.
  - Stacks: LIFO
  - Lists: Ordered
  - Sets: Unordered, unique
  - Maps

Search Trees (BST, B, LLRB)

- Depth = dist from root (depth of root = 0); height = max depth (gives worst case)
- BST
  - BST for Sets : $\Theta(\log N)$ if balanced; Hibbard deletion.
  - Insert randomly into a BST, then the average depth and height are expected to be $\Theta(\log N)$

  - All left nodes are smaller; all right nodes are larger.

- B-Tree (2-3 Tree): always balanced, slow and complicated – Balanced, $\Theta(\log N)$ for contains() and add() given constant L for node limit
  - Inv 1: All leaves must be the same distance from the source
  - Inv 2: A non-leaf node with k items mut has exactly $k+1$ children.
  - height always between $\sim \log_2 N$ and $\sim \log_{L+1} N$
- LLRB (rotation): 1-1 correspondence with 2-3 trees – Balanced, $\Theta(\log N)$ for contains() and add()
  - No node has 2 red links.
  - Every path from root to leaf has same number of black links (because 2-3 trees have same number of links to every leaf).
  - There are no red right-links.
  - For height H corresponding 2-3 tree, LLRB height $\leq 2H+1$, so max height still $\Theta(\log N)$
  - Adding: add a leaf with red link, and then do rotation, flipping etc.

Hashing, Heaps

- Hashing
  - Consistency: when hashCode() is called on the same object multiple times, the same value should be returned.
    Subtle issue: the items being hashed should NOT be mutable, or else we might not be able to find the item after changing – it's in the original hashCode bucket, but will "disappear".
  - Equality: if o1.equals(o2) == true, then hashCode(o1) should be equal to hashCode(o2).
    Never override equals() without also overriding hashCode()
  - NO duplicate (keys), so contains() and add() are both $\Theta(Q)$ where Q is the length of the longest list. N.B. HashMap will overwrite values with the same key
  - Valid iff returns integer and equals() items return the same hashCode (deterministic).
  - Load factor = # of items / # of buckets – do resizing
  - Use a base of small prime # (e.g. 31) to guarantee uniform distribution of bins after mod.
  - Worst Case add() happens during resize
  - HashMap method: containsKey
  - hashCode & 0x7FFFFFFF to set the sign bit to 0 → non-negative
    (hashCode & 0x7FFFFFFF) % arr.length;
  - String and other Object subclass calling super.hashCode() will return the memory address, the Object class hashCode().
  - hashCode of "ab" $= 'a' \cdot 31 +' b' = ('a'-1)\cdot 31 + ('b'+31)$
  - Extra: $256^k == 0$ for any int $k \geq 4$
- (Heap and) Priority Queue : Optimizes handling min/max element(s) ⟶ Binary Min-Heap (swim/sink), inserts top to bottom & left to right
  - Runtime: add, removeSmallest: $\Theta(\log N)$; getMin: $\Theta(1)$
  - Min-heap Property: Every node $\leq$ both children

- Complete: Missing items only at bottom level, all nodes as far left as possible
- Implementation: 1 empty spot in the beginning of the array (leftChild $2k$, right $2k+1$, parent $k/2$) (less memory usage than tree repr)
- MinHeap to MaxHeap : Insert the negation of number, and negate again when removing
- PQ $\rightarrow$ Stack (LIFO) & Queue (FIFO)

Tries, K-d Trees

- Trie
  - Runtime: add() and contains() both: $\Theta(L)$ where L is the length of the key
  - Invariant: each node stores one character
  - Impl: DataIndexedCharMap: fastest, too much memory; BST: better
  - Impl: HashMap<Character, Node>: typical choice
  - Extra : Each node stores its own value as well as the value of its best substring (PQ)
- Multi-dimensional data
  - Uniform Partitioning – still $\Theta(N)$
  - QuadTrees – 4 neighbors of NE, NW, SE, SW
  - K-d Tree : Split along x, y alternatively; always draw two (empty/null) branches when the node is set up
    - Worst-case runtime for closest point (aka nearest neighbor) search in a reasonably bushy k-d tree $\sim \frac{1}{2}N = O(N)$
    - Optimization: Do not expolore subspaces that aren't possible to make it better.
      Process: Go through the better side first before deciding whether we could drop the worse side or not, i.e. update minDistance after going through the good side would prune bad side more often
    - Range finding & Nearest neighbor

Tree Traversals, Graphs

- Traversals
  1. Level order traversal.
  2. Depth-First traversals – of which there are three: pre-order (root LR), in-order (L root R) and post-order (LR root).
- Only consider simple graphs (no double edges and self-loops), but can be either directed or undirected.
- DFS (depth-first traversal) and BFS – don't care about weights – $O(E+V)$ time and $O(V)$ space
  - DFS is worse for spindly graphs (recursive or iterative implementation with LIFO fringe, i.e. Stack)
    - Usage: cycle detection
    - Post-order DFS reversed = Topological sort
      Rules: all parents of v appear before that v; can only be done on a directed, acyclic graph.

- BFS is worse for "bushy" graphs; finds shortest paths in an unweighted graph (iterative implementation with FIFO fringe, i.e. Queue)
  - For two vertices x, y simultaneously on the FIFO queue (fringe) at some point during the execution of BFS from s, then the len of the SP between s, x is at most 1 more than the len of the SP between s, y
  - Usage: social network, GPS search for nearby hotels etc.
- Adding a constant positive integer k to all edge weights will affect some shortest path between vertices; while multiplying ($\mathbb{R}^+$) won't

- Dijkstra's (SPT from source s)
  - Runtime
    - add : V, each costing $O(\log V) \rightarrow O(V \log V)$
    - deleteMin : V, each costing $O(\log V) \rightarrow O(V \log V)$
    - changePriority : $O(E)$, each costing $O(\log V) \rightarrow O(E \log V)$
    - Overtall : $O(V \log V + E \log V)$; assuming $E > V$ (usually, e.g. connected graph), then $\rightarrow O(E \log V)$
    - Array-based Queue implementation: add $O(1)$, delMin $O(V)$ – faster when graph is dense, i.e. $E \sim V^2$
  - Invariants
    - edgeTo[v] is the best known predecessor of v
    - distTo[v] is the best known total dist from source to v
    - PQ contains all unvisited vertices in order of distTo
  - Properties
    - Always visits vertices in order of total distance from source
    - Relaxation always fails on edges to visited vertices (even if better path due to negative weights)
  - Guaranteed to be optimal (without negative edges)
  - If there was any point where there is a positive edge weight between a negative edge weight and the s, it is possible that Dijkstra's would mess up. This only works if the graph has no negative cycle, else we would get an infinite loop.

- A*
  - If only a single target in mind, then A* over Dijkstra's
  - Adding a hypothesis (shortest edge outwards); Estimate : arbitrary heuristic : using experience to learn and improve - doesn't have to be perfect!
  - Go to the next vertex with smallest sum of inward ($\sim$ Dijksta's) + outward
  - For A* to be correct, the A* heuristic must be (1) Admissible and (2) Consistent
  - A heuristic is admissible if all of its estimations $h(x)$ are optimistic, i.e. $h(v) \leq$ the true distance from s to v
  - 1. Choice of heuristic matters; 2. A bad choice can be bad
  - If all heuristics are the same, then equiv. to Dijkstra's $\rightarrow$ worst case $O(E \log V)$ assuming heuristic calculations are $O(1)$

- Minimum Spanning Tree (MST)

- Cut Property : given any cut, the min weight crossing edge is in the MST; justifies the 2 MST methods below
- Prim's and Kruskal's algorithm output a minimum spanning tree for connected and undirected graph.
- Prim's – engulfing nodes with a starting node, keeping dist from unmarked nodes to the MST under construction
  - Using a PQ fringe, assuming all PQ operations take $O(\log V)$,
  - PQ add: $O(V \log V)$
  - PQ delMin: $O(V \log V)$
  - PQ relax: $O(E \log V)$
- Kruskal's – picking edges with weights in ascending order
  - Assuming all WQUPC operations take $O(\log^* V)$
  - PQ add: $O(E \log E)$
  - PQ deleteMin: $O(E \log E)$ without being pre-sorted; $O(E)$ with all edges pre-sorted
  - SQUPC union: $O(V \log^* V)$
  - WQUPC isConnected: $O(E \log^* V)$
- Unique given distinct weights
- For many graphs, MST $\neq$ SPT for any particular vertex
- Adding a positive constant to all edges or multiplying a positive constant does NOT affecting the MST.
- A graph's longest edge can belong to the MST
- A min-weight edge on some cycle might NOT belong to the MST

- Cycle Detection : If the graph is directed, then there is a cycle if a node has already been visited and it is still in the recursive call stack (i.e. still in the fringe and not popped off yet). Assuming an adjacency list implementation, the runtime of this is equivalent to the runtime of depth first search, which is $\Theta(|V| + |E|)$.
- The shortest edge in any cycle is NOT always be a part of a MST
- Maximum Spanning Tree – negate all weights and run Prim/Kruskal

Extra Sanity Checks

- If want to find median, quartiles, etc., keep two Min/MaxHeap(s); If want to delMin + delMax, Trees (LLRB, BST etc.)
- Can never call a method on or check an instance variable of left or right without first doing a null check
- Java features (61B)
  - Packages.
    - Good: Organizing, making things package private
    - Bad: Specific
  - Static type checking.
    - Good: Checks for errors early , reads more like a story
    - Bad: Not too flexible, (casting)

  - Inheritance.
    * Good: Reuse of code
    * Bad: "Is a", the path of debugging gets annoying, can't instantiate, implement every method of an interface
  - Shortest Paths
    * Dijkstra's : Find smallest unvisited vertex (PQ) and relax all edges; works for $\mathbb{R}^+$ weights
    * A^* : Optimizes for 1 target by adding a heuristic estimation

- When asked about runtime: Master Theorem OR if given a chart of runtimes v. sizes, assume aN^b and estimate
- The height of a leaf node in a rooted tree is 0
- For MinHeap, given any MinHeap with N distinct keys, the result of inserting a (max) key ($>$ any of the N keys), and then deleteMax() does NOT necessarily yield the original MinHeap.
- $\Theta(c^{\log N}) = \Theta(N)$; $\Theta(\log N^c) = \Theta(\log N)$; $\Theta(\log N!) = \Theta(N \log N)$
- Use .equals() and .compareTo() for Strings etc.
- EXTRA: Each ADT: interface + methods (best, worst case runtime + respective base implementation)
  - DisjointSets (no efficient deletion)
    * Interface
      · void connect(int a, int b);
      · boolean isConnected(int a, int b);
    * Runtime (constructor, connect, isConnected)
      · ListOfSets: $\Theta(N), O(N), O(N)$
      · QuickFind: $\Theta(N), \Theta(N), \Theta(1)$
      · QuickUnion: $\Theta(N), O(N), O(N)$
      · WQU: $\Theta(N), O(\log N), O(\log N)$
    * Usage: cycle detection (Kruskal's)
  - BST
    * Our Interface
      · search, insert, delete (If 2 children then do Hibbard deletion, i.e. pick right-most from leftChild or left-most from rightChild and make it the new root)
      · int size(); (etc. Shared by all $\in$ Collections)
    * Runtime (constructor, search, insert, remove)
      · average: $O(N \log N), O(\log N), O(\log N), O(\log N)$
      · worst: $O(N^2), O(N), O(N), O(N)$
      · Height: $\Omega(\log N), O(N)$
  - B-Tree, LLRB − insert, remove, and height are $\Theta(\log N)$; search $\Omega(1), O(\log N)$
  - Hash Table (Map)
    * Java Interface
      · add(E e) : amortized $\Theta(1)$, worst case (resize) $\Theta(N)$
      · contains(E e) : amortized $\Theta(1)$, worst case (bad hashCode) $\Theta(N)$

        - HashMap: containsKey(K k), containsValue(V v), keySet(), put(K k, V v)
    - Heap (PQ - unordered array, ordered array, minHeap) − complete binary tree
        - Java Interface
            - getMin(); − $\Theta(1)$, root
            - insert(E e); − $O(\log N)$, add to bottom level right-most pos, and swim up.
            - removeMin(); − $O(\log N)$, move the bottom level right-most element to root and sink down.
        - Construction: store keys in array that represent the heap structure since complete, with 1 empty in front
            - parent: $k/2$
            - left/right child: $2k, 2k+1$
    - Trie
        - add() and containts() both $\Theta(L)$ where L is the length of the word, i.e. if constant len, then $\Theta(1)$
        - Supports prefix matching efficiently
    - K-d Tree − BST in disguise, swapping dimensions at each level
    - Graph
        - Integer vertices : HashMap<String, Integer> with # of vertices specified in advance
        - Impl of E and Runtime of graph printing:
            - Edge Sets: HashSet<Edge>, where each Edge is a pair of Int's, $\Theta(V^2)$
            - Adjacency matrix: bad due to $\Theta(V^2)$ space
            - Adjacency lists: Array of lists, $\Theta(V+E)$ (typical since most graphs are sparse)

- Usually goes for delegation not extends
- For a collection of N reviews, to determine the number of reviews within a certain date-time range.

  Use ArrayList<Review>, so construction is $\Theta(N \log N)$, and query (endpoints) is $\Theta(\log N)$
- Given N images of size 256x256, each represented as a 2-D Array (i.e. int[][]). Each image has associated with a saturation value, which can be calculated from the pixels. Support add(int[][] img), getAllImgWithSaturation(int saturation), and remove(int[][] img).

  Use HashMap<Integer, HashSet<ImageContainer>> with helper class ImageContainer, so construction is $\Theta(N)$
- The golden rule for static methods to know is that static methods can only modify static variables.