Sorting

~~~ Sequence ~~~

//Sequence after bubble/selection/insertion sort n times with n+1 term:

Sort entire sequence.

//Sequence after insertion sort n times:

Sort from first to (n + 1)th term.

//Sequence after selection sort n times:

Obtain manually.

~~~ Passes ~~~

//Bubble sort number of passes:

If end early, add 1 to passes due to checking round.

~~~ Swaps ~~~

//Swaps required for bubble sort:

Count manually.

~~~ Comparisons ~~~

//Comparisons for insertion sort:

For every term starting from the second term, += count until (inclusive) the first term smaller than it.

//Comparisons for bubble sort:

If n integers, answer = n-1 + n-2 + ... until finish sorting and last round of check.

E.g. 6 integers requiring 2 passes, answer is 5 + 4 + 3 = 12.

~~~ Pivot ~~~

//Quick sort possible pivots:

For every term, check if there is larger term on left or smaller term on right.

Linked List

~~ Pseudocode ~~

//Output:

Count by keeping track of n position.

~~ Peek ~~

//Stack:

Peek at tail.

Push & pop at tail.

//Queue:

Peek at head.

Push at tail, pop at head.

Note: If no. of dequeue < original size, ignore them.

Binary Max Heap

~~~ Max Heap Property ~~~

//Subsets that violates max heap property:

Select nodes where not all nodes under it (not just the child) are smaller than it.

//Subset that is less than n:

Select all nodes less than n.

//Max heap validity:

Check if all parents have child that are smaller or equal.

~~~ Comparisons ~~~

//Maximum comparisons for O(n) build heap:

For all non-leaves, count number of comparisons in longest (left-most) path to bottom.

Note: At index n, count++ for each child (up to 2), then set left child as next n.

//Minimum comparisons for O(n) build heap:

Count number of branches.

~~~ Swaps ~~~

//Maximum swaps for O(n) build heap:

For all non-leaves, count number of left child to reach bottom.

//Sequence of vertices that will swap by inserting n:

Identify starting location, then move upwards.

~~~ Extraction ~~~

//Select remaining nodes after n extraction:

Count total to determine number of remaining, then select that number of smallest.

Hashing

~~~ Sequence ~~~

//Probe sequence when deleting n:

Identify base index, then increase step until arrive at n.

//Sequence when linear probe with hash function is h(key) = key mod m:

Obtain m value from number of cells. Choose any value, select if valid until all selected.

~~~ Validity ~~~

//Validity of result after insertion:

Use tool to check.

Binary Search Tree

~~~ Structure ~~~

//Click the root of this BST:

Select the root.

//What is the height of this BST:

Root is at height 0, count to leaf.

//What is the value of element with rank n in this BST:

Get the nth smallest element.

//Click all the internal vertex:

Select all nodes that are neither leaf nor root.

//What is the minimum/maximum element:

Select the node with smallest/largest value.

//How many structurally different BSTs from n distinct elements?

(2n)! / ((n + 1)! * n!)

0 1 2 3 4 5 6

1 1 2 5 14 42 132

~~~ Search ~~~

//Click the sequence of vertices visited by Search(n):

From root to node n, select all visited.

~~~ True or False ~~~

//Insert operation is always not commutative:

False

//It is possible to have a search sequence is as follows:

If current value larger, next value must be smaller and vice versa.

//Smallest element has no predecessor/left child:

True

//Smallest element always has a successor:

False

//Smallest element always has no right child:

False

//Largest element always has a parent:

False

//Largest element always has no right child:

True

~~~ Successor/Predecessor ~~~

//Click the sequence visited by Successor/Predecessor(n):

From n (inclusive), select all nodes visited until its successor/predecessor (inclusive)

~~~ Validity ~~~

//Is the graph shown a valid BST:

Check if any vertex violates BST property.

~~ Preorder/Postorder ~~~

//Click the sequence of vertices that are visited by preorder(root):

From root, go to the left-most value, then repeat {clear left child, clear right child and move up}

//Click the sequence of vertices that are visited by postorder(root):

From left most value, move right by clearing all left child, then right child, before moving up to its parent, until root.

Note: Don't include parent as visited until all its child are clear.

AVL Tree

~~~ Height ~~~

//What is the minimum number of vertices with height h:

N(h) = n(h -1) + n(h-2) + 1

0	1	2	3	4	5	6	7	8	9	10	11	12
1	2	4	7	12	20	33	54	88	143	232	376	609

~~~ Delete & Insert ~~~

//Delete at least n vertices so that n rotations:

If n = 0, delete from leaves.

If n > 0, guess and check from bottom layer.

//An integer n is to be inserted; will it result in any rotation:

Identify position and check if unbalanced.

//Will the deletion of n result in any rotation:

Remove and check if unbalanced.

Graph

~~~ AM/AL/EL ~~~

//How many entries are there in edge list:

If directed, count edges where bi-directional as 2 entries.

If undirected, each bi-directional is 1 entry.

//Which best graph when existence of edge(u,v) is asked frequently:

AM

//Which best graph when neighbors are frequently enumerated:

If vertices^2 <= max, AL and AM

If vertices^2 > max, AL only

//Which best graph when edges need to be sorted:

EL

//How many cells are there in AM:

Square of number of nodes.

//How many cells in the AM are non-empty/zero:

Count number of edges, x2 if bidirectional.

//What is the value of adjMat[n][m]:

If n = m, 0.

Else if there is an arrow from n to m, answer is weight. Otherwise, 0.

//Select entries that belong to AL of graph:

Check neighbors and ensure no repetition.

//Click on all neighbors of n:

Select all adjacent.

~~~ Drawing ~~~

//Draw a complete graph with n vertices:

Draw n vertices (note: stop at n-1), then have an edge between all of them.

//Draw a tree with n vertices:

Draw n vertices (note: stop at n-1), then connect to form a tree.

//Draw a directed acyclic graph with n vertices and m edges:

Draw n vertices (note: stop at n-1) and m edges, ensure no cycle.

~~~ Degree ~~~

//What is the sum of degrees:

Count number of edge, x2 if bidirectional.

//What is the in/out-degree of vertex n:

Count number of edges going in/out of n.

~~~ Structure ~~~

//Select statements that are true:

Determine if weighted, directed and cyclic.

Graph Traversal

~~~ Paths ~~~

//Select options that represent valid simple paths:

Select all that are possible and do not have repeated nodes.

//In the following DAG, how many simple paths are there from n to m?

Run through all cases.

~~~ Spanning Tree ~~~

//Click all edges that must belong to every spanning tree of the graph:

Select edges that lead to nodes with only 1 edge.

~~~ Deletion ~~~

//Click subset of vertices that cause the graph to be disconnected if deleted:

Select all where removing just that one node would disconnect the graph.

//Suppose vertex n is removed, how many connected components are there:

Count number of components without n.

~~~ DFS & BFS ~~~

//Click all edges that make up spanning tree by BFS starting from n (ascending):

From n, identify smallest neighbor and add to queue. Exclude edges that lead to visited node.

//Click all edges that make up spanning tree by DFS starting from n (ascending):

From n, go through the first path, then backtrack till have unvisited neighbor and continue.

//DFS and BFS can run in O(V^2) on:

DAG, Bipartite and Complete

//DFS and BFS always run in O(V^2) on:

Complete

//Click the sequence of vertices visited by BFS from n:

From n (inclusive), identify smallest neighbor and add to kill.

//Consider pseudocode for DFS, click sequence.

If print is before for loop: Starting from root, clear subtree from left to right.

If print is after for loop: Starting from left-most subtree, repeat {check left child, right child then parent}

~~~ Toposort ~~~

//Click sequence of vertices that result in a valid toposort:

Starting by selecting all nodes with in-order = 0, then select its children.

//Is [] a valid toposort of the graph:

Check all from first, ensure that the incoming has been visited.

~~~ Bipartite ~~~

//Is the graph bipartite:

If tree, yes. Else, if there are any shapes with odd number of edges, no.

//Draw a simple bipartite graph with n vertices and m edges:

Draw n vertices that separate into 2 rows, then draw edges across rows.

Note: Up to 1 edge between 2 nodes.

Single Source Shortest Path

~~~ SSSP in general ~~~

//What is the weight of the shortest path from n to m:

Check all paths and identify the shortest.

//Run <u>original</u> Dijkstra's algorithm and Bellman-Ford's algorithm from source n:

If negative weight cycle unreachable from n, can get correct shortest path. Else, both cannot.

Both can always terminate.

//Run <u>modified</u> Dijkstra's algorithm and Bellman-Ford's algorithm from source n:

Bellman-Ford can always terminate.

Modified Dijkstra can only terminate if reachable nodes do not result in negative weight cycle.

Both will give wrong answer if reachable node results in negative weight cycle.

//Click the sequence of vertices that constitutes shortest path from n to m:

Check all paths and identify the shortest.

//Click the subset of vertices reachable from node n and have shortest path less/greater than m:

Select all visited from n within or more than m weight.

~~~ Bellman-Ford's Algorithm ~~~

//Click the sequence edges are relaxed using One-Pass Bellman-Ford's algorithm:

Select in toposort order.

//Source vertex q. After n pass of the Bellman-Ford's algorithm, what is the value of D[m];

Starting with inf for all except for source vertex = 0, run through the order of edge n times.

Ignore all the edges where nodes have not been “activated” yet.

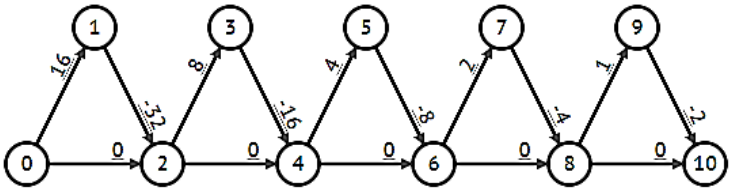
//Draw a simple connected weighted … Optimized Bellman-Ford’s … at least n rounds to get shortest path:

Have nodes in reverse order in a chain, except for 0 which is all the way in front.

~~~ Dijkstra's Algorithm ~~~

//Draw a simple connected weighted … modified Dijkstra algorithm … successful relaxes >=q edges:

Follow the diagram but ensure that edges are less than maximum. (remove negative weighted from right side)



//Draw a simple connected weighted … modified Dijkstra's algorithm … run indefinitely:

Include a negative weight cycle. (Large negative value) All weight must be distinct.

//Run modified Dijkstra's algorithm. Does it always produce shortest path for all values:

If have negative weight cycle, no. (Take note of bi-directional ones)

~~~ BFS ~~~

//Run BFS on the graph from node n. Does it produce shortest path for reachable:

Yes, only if all reachable vertex forms a tree or unweighted graph. (No cycle)