Exam 1

For determining big-Oh of functions, first simplify logs and exponents. Refer to the logarithms and exponents reference (posted on the schedule page) for useful rules such as

$$b^{log_b k} = k$$

and

$$log_b n^k = k log_b n$$

For ordering, compare powers of the higher-order components of the product first. For functions of the form $n^c \log^d n$, order first by c and then, for the same values of c, by d. The functions from 1a from slower-growing to faster-growing: n, $n \log n$, $n \log^3 n$, $n^{1.5}$, n^2, $n^3 \log n$, $n^6 \log n$.

Running times should be expressed in terms of the inputs to that thing. `helper` in both #2a and #2b takes an array of length m and a value k, so the running time should be expressed in terms of k and/or m. ($O(k)$ in both cases.)

Typo in #2a — the middle loop should start with $i \leftarrow; 1$ rather than 0 since `i *= 2` will be 0 when i is 0 and thus the loop would never exit.

The best case is *not* small n; it is the least work for a given size of n. For #2a, the best case for the middle loop is if `A[i] = 0` on the first iteration — if so, the loop immediately exits ($O(1)$). The worst case is if `A[i]` is never 0. For #2b, there's only one route through `alg` for any particular n so the best and worst cases are the same; the base case ($n < 4$) involves less work than the recursive case ($n \geq 4$) but that is for different values of n.

The middle loop in #2a repeats for $i = 1, 2, 4, 8, \ldots, n$. This is $\log n$ repetitions, but to say that the total work is the number of repetitions times the worst-case work in the body ($O(i)$ for $i = n$) overcounts — since the work is different for each repetition, write the sum.

`alg` in #2b is recursive — write the recurrence relation. How many recursive calls are there? What is the size of the problem in each call? How much other work is done in `alg` other than the recursive calls themselves? As part of the latter, note that while `helper` is $O(k)$, it is only ever called with $k = 4$ so each call is $O(1)$.

$f(n) = O(g(n))$ means that g is an upper bound on f — f grows no faster than g (and it might grow slower). $f(n) = \Omega(g(n))$ means that g is a lower bound on f — f grows no slower than g (and it might grow faster). $f(n) = \Theta(g(n))$ means that g and f grow

at the same rate.

For #3a, (i) can be true because $f(n) = \Theta(n)$ means that $f(n)$ grows at the same rate as n and $f(n) = O(n \log n)$ means that $f(n)$ grows no faster than $O(n \log n)$ — these are compatible statements. (ii) is not true because $f(n) = \Omega(n \log n)$ means that $f(n)$ grows no slower than $n \log n$ and that is not compatible with $f(n)$ growing no faster than n.

O does not mean "worst case" and Ω does not mean "best case". Best- and worst-case refer to the behavior of algorithms and to the least and most work, respectively, for a given n. There is no best or worst case for a function; it's just a function. Where the concepts intersect is that O establishes an upper bound on the growth rate of a function while the worst case establishes an upper bound on the time the algorithm takes on any input of size n. Ω is similar, establishing a lower bound on the growth rate of a function while the best case establishes a lower bound on the time the algorithm takes on any input of size n.

For #3b, "an algorithm taking $O(n \log n)$ time" covers all inputs of size n — the running time of the worst case is $O(n \log n)$. (i) is not true because $\Theta(n^2)$ says the running time for those inputs grows like n^2, but n^2 grows faster than $n \log n$ and so is not compatible with the algorithm's running time being $O(n \log n)$. (ii) is true because a lower bound on the running time of the best case gives a lower bound on the running time of every input and "growing at least as fast as $n \log n$" is compatible with "growing no faster than $n \log n$". (iii) is true because "grows like n" and "grows no faster than $n \log n$" are compatible statements.

Exam 2

For (regular) binary search trees (not necessarily balanced), while it is technically correct to say that find, insert, and remove are $O(n)$ because the worst-case height of the tree is n, it is more informative to say that the operations are $O(h)$ where the height h of the tree is $O(n)$.

#1 asked for worst-case running times. For a hashtable with separate chaining, find and remove are worst-case $O(n)$ because all of the elements could hash to the same slot and the element in question might be at the end of the list for that slot. Insert is worst-case $O(1)$ because the new element is added at the head of the list so it doesn't matter how many elements are in the list. The $O(1)$ time we think of for hashtables is only *expected* time (on average, in a probabilistic sense) when the load factor is low. For separate chaining it is more informative to say that find and remove are $O(n/N)$ expected where n is the number of elements in the hashtable and N is number of slots in the hashtable — there should be approximately n/N elements in each slot if the

hash function does a good job of distributing keys.

#2 and #3 are each asking you to design a single thing to support all of the operations listed, not what would be best to support each operation in isolation.

Keep maintenance costs in mind — if you have multiple data structures to support different kinds of fast access, you also have to keep them all updated when things are added to the collection.

Removing an element from a collection typically involves first locating the position of the element within the collection — the index in the array, the node in a linked list or tree, etc — before updating the structure to do the actual removal. If finding the element is more expensive that the removal once found, keep in mind how we bypass search: do lookup instead. Again, keep maintenance in mind — if you store references to positions, those references have to be updated any time an element's position changes.

A hashtable stores key-value pairs. If you are using a hashtable to store things, be sure to identify both the keys and the values. "A hashtable storing session IDs" is incomplete — if you only have session IDs and are using the hashtable for membership questions rather than lookup of values, use a set instead.

Similarly, heaps and binary search trees are ordered — saying just "heap" or "BST" is incomplete, instead identify "heap ordered by ...".

Terminology: "searching" suggests going through multiple elements to locate something whose position is unknown. Hashtables are for lookup, where the position of the key is known (or can be easily computed) and you just want to retrieve what is there.

(Balanced) binary search tree operations are only $O(\log n)$ if you are searching on the same field the BST is organized on e.g. searching for a session ID in a BST organized by session IDs. If your BST is instead organized by the last activity timestamp, it is unsorted with respect to session ID and searching the BST for a particular session ID requires full traversal.

For #3, store the last activity timestamp and compute the idle time as the current time minus the last activity time (as the problem says). Storing the idle time directly is problematic because it increases for all sessions as time goes by.

`closeIdle(timeout)` should close all k sessions where the idle time exceeds the timeout. The required running time of $O(k \log n)$ means that you need to be able to locate all the sessions to close without simply going through all of the sessions. How can you organize the sessions so that the first k are the ones to close?

Exam 3

For #1, list the MST edges in the order they are found. Kruskal's algorithm considers all edges in order of increasing weight, adding the current edge to the MST if it doesn't form a cycle. Prim's builds the spanning tree from a single starting vertex, adding the cheapest edge connecting a new vertex to the tree being built. This can be done by maintaining a priority queue of edges ordered by weight, but it is more common to have a priority queue of vertices ordered by the cheapest-weight edge connecting that vertex to the tree-so-far.

Scenario A is not simply a shortest weighted path task — how can you build a graph so that taking into account the locked doors and a limited number of keys? Modifying Dijkstra's algorithm is not the solution here, instead, build a graph so that running a normal Dijkstra's results in the correct answer.

For scenario D, both "similar" and "opposite" denote a relationship between tweets, but they have very different meanings. You want two groups of tweets, which sounds like 2-coloring — do you want "similar" or "opposite" edges in order for 2-coloring to produce the desired groups?