Problem 1. (a) Ten people is too large a group (violating part 3 of the collaboration policy). It would be better to split into groups of 3-4.

(b) Ambiguous - ideally each person should independently take notes of the sketch, rather than taking photographs of the same work (especially so if the sketch is detailed).

(c) This violates part 1 of the collaboration policy.

(d) Ambiguous - depending on how you go about pointing out the flaw, it could turn out to be a cooperative effort or reduce to you telling the other person the solution.

(e) I think this is okay. The other person is still independently figuring out their solution and writing it up, albeit with a slight bit of help (as long as the other person does not take pictures or write down detailed notes of your writeup).

(f) I think this violates part 1 of the collaboration policy, since exchanging writeups and correcting each others' solutions is not working independently.

(g) This violates the collaboration policy regarding seeking other sources for solutions.

(h) Since you cannot "forget" a solution, this situation is a bit difficult. That being said, you can still think of other ways to solve the problem (at least to a reasonable extent) before committing to the solution in your mind.

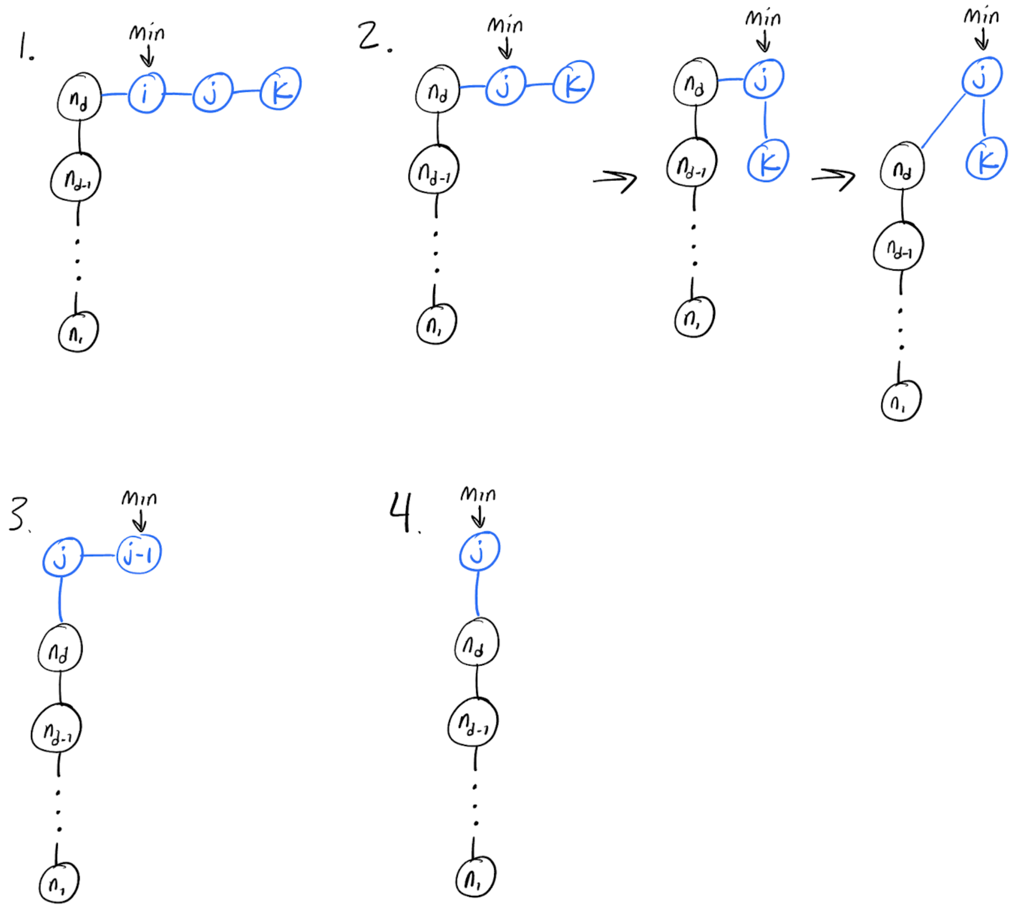
(i) This is an extension of (g), even if you are the one explaining the solution - it comes from another source, which is not allowed by the collaboration policy.

Problem 2. We'll prove that given a Fibonacci heap that consists of a heap-ordered tree that is a chain of d nodes, we can perform a sequence of operations that produces a HOT of $d + 1$ chained nodes as follows:

1. Insert 3 items $\{i, j, k\}$, where $i < j < k <$ `heap-min`.
2. Delete-min.
3. Decrease-key of k to $j - 1$.
4. Delete-min.

The first step appends the newly-inserted items to the list of roots and sets i to be the `heap-min`. The second step deletes the `heap-min` (i.e. i), sets j as the `heap-min`, and consolidates, which first merges nodes j and k, and then merges j and the original HOT of d chained nodes, leaving a single tree with root node j and children k and the original HOT. The third step cuts k, decreases its key to $j - 1$ (the new `heap-min`) and appends it to the list of roots. Finally, the fourth step deletes the `heap-min` (i.e. k), leaving the Fibonacci heap as a single HOT of $n + 1$ chained nodes.

Starting from depth $d = 0$ and building up (down) the chain, we find that after performing operations on n nodes we are left with a HOT of $n/3 = \Omega(n)$ chained nodes.



Problem 3. Todo.

Problem 4. (a) We can create a data structure that consists of a structure like P and a linked list L. We then define the following operations (using `delete-min`$_\text{P}$, `merge`$_\text{P}$, `make-heap`$_\text{P}$ to refer to the operations of P):

- `Insert`: append the item to the linked list L (we maintain a pointer to the tail).
- `Delete-min`: turn L into a temporary priority queue P_{temp} using `make-heap`$_\text{P}$, merge P_{temp} and P using `merge`$_\text{P}$, and delete the minimum using `delete-min`$_\text{P}$. At the end, reset L to an empty list.
- `Merge`: for both data structures, turn L_1 and L_2 into temporary priority queues P_{temp1} and P_{temp2} using `make-heap`$_\text{P}$, merge (P_{temp1} and P_1) / (P_{temp2} and P_2) using `merge`$_\text{P}$, and finally merge the two resulting priority queues. At the end, reset L to an empty list for the current data structure.

The runtime analysis is as follows:

- `Insert`: Appending to the tail of the linked list costs $O(1)$ time. Additionally, later on when `delete-min` and `merge` are called, each item costs $O(1)$ future time when L is turned into a priority queue during those operations. Therefore, the total amortized cost of insertion is $O(1)$.
- `Delete-min`: The cost of turning L into a temporary priority queue during this operation is already paid for by the amortized cost of insertion, which leaves the real cost of calling `merge`$_\text{P}$ and `delete-min`$_\text{P}$, both of which are $O(\log n)$ amortized time, and resetting L, which is $O(1)$. Therefore, the total amortized cost of this operation is $O(\log n)$.
- `Merge`: Similarly, turning L into a priority queue during this operation is already paid for by amortized insertion, which leaves the real cost of calling `merge`$_\text{P}$ three times and resetting L, giving a total amortized cost of $O(\log n)$ for this operation.

(b) We can create a heap-of-heaps data structure, where the main heap H_{main} consists of a collection of sub-heaps keyed by their root elements such that the sub-heap containing the minimum element is the main heap's "root". We use the priority queue black box as our heap implementation, and like before, we also maintain a linked list L. We then define the following operations (again using `operation`$_\text{P}$ to refer to the operations of P):

- `Insert`: append the item to L.
- `Delete-min`: turn L into a heap using `make-heap`$_\text{P}$ and insert the resulting heap H_{new} (keyed by its minimum element) into the main heap H_{main} using `insert`$_\text{P}$. Then call `delete-min`$_\text{P}$ on H_{main} to get the sub-heap H_{min} containing the minimum element of the entire data structure, and then call `delete-min`$_\text{P}$ again on H_{min} to delete the minimum element. If H_{min} is not empty at this point, re-insert it into the H_{main} using `insert`$_\text{P}$. Finally, reset L to an empty list.

Runtime analysis: for `insert`, appending the item to L costs $O(1)$ time and also incurs $O(1)$ future cost during `make-heap`$_\text{P}$; therefore the total amortized cost for insertion is $O(1)$. For `delete-min`, the cost of turning L into H_{new} is covered by the amortized cost of inserting elements, which leaves the real cost of 2 `delete-min`$_\text{P}$ operations and 1-2 `insert`$_\text{P}$ operations, all of which are $O(\log n)$ amortized time (in the worst case, both H_{main} and H_{min} can contain at most n items), and resetting L, which is $O(1)$. Therefore, the total amortized cost for deleting the minimum of the data structure is $O(\log n)$.

Problem 5. (a) We adapt the approach shown in class for planar point location using persistent trees for this problem. Imagining the number line as the time axis, each of the closed intervals $[a_i, b_i]$ define an interval of time. We can visualize these time intervals as horizontal line segments in 2D space, where each segment gets its own "lane" so as not to overlap with another segment. Then, we can partition the space into vertical slabs based on the endpoints of the segments. The question thus becomes determining the number of line segments within the slab that contains the query point x.

Using a persistent binary search tree (e.g. a red-black tree) as our data structure, we can scan across the number line and update the tree. Passing through the slabs, whenever we encounter the start of a time interval, we add its y-coordinate to the tree at the current "time" x-value, and whenever we encounter the end of a time interval, we delete the node from the tree. After finishing the scan, we can use binary search to pinpoint the time-slab that contains the query point x, and return the intervals contained in that slab (i.e. the nodes of the tree at that point in time).

During the construction of the persistent BST, inserts and deletes take $O(\log n)$ time and there are $O(n)$ such operations for n time intervals throughout the course of the scan, so the persistent tree is constructed in $O(n \log n)$ time. A persistent red-black tree uses $O(n)$ space due to the fact that we don't care about the old values of red/black bits and only persist the current values and rotations. Lastly, finding the time-slab using binary search is $O(\log n)$, and traversing the tree with k nodes to output the intervals present is $O(k)$. Therefore, the total runtime is $O(k + \log n)$.

(b) Given that the interval endpoints are contained in the range $1, \ldots, n$, we can use a persistent linked list to keep track of the active intervals. A node in a persistent linked list is a pointer-based structure with a constant indegree of 1 and a constant number of fields pointing to the next node in the list, so updates (insertion and deletion) cost $O(1)$ time. Since there are n intervals, the total construction time for the persistent linked list structure is therefore $O(n)$, and the space used is also $O(n)$. Querying for a specific time is constant, since we can store the pointers to the roots in a pre-defined array of length n and lookup the pointer in $O(1)$. Once we have the root at time x, all that remains is to traverse the linked list and output the present intervals. Thus, the total query time is $O(k)$.