PS6

A – Multiples of 3

- (The detailed algorithm is quite complicated so this diary entry only outlines the key milestone when I was working on the problem)
- Range tree structure:
    - First by reading this question, the tree structure is straightforward as the number of values who are multiples of 3 at a node is the sum of number of values who are multiples of 3 at its children. So intuitively, each node should store the number of values which are multiples of 3
- Updating:
    - The issue I encountered with updating was, if we increase all values in a particular range by 1, then what happen to the number of values who are multiple of 3? We cannot conclude the answer straight up, but we can observe that there exist only 3 possible changes:
        - The values who are originally multiples of 3, now are no longer multiple of 3
        - The values who are 2 more than multiple of 3 becomes multiple of 3.
        - The values who are 1 more than the multiple of 3 becomes 2 more than multiple of 3
    - The behaviour above reminds me of mod operation. Hence I made each node into struct with 3 attributes, the number of values that is 1mod3, 2mod3, 0mod3 respectively.
- Lazy counter:
    - This is where the lazy counter comes into play. Since the mod values are simply cycling around with period = 3, we can store modulo of 3 for the lazy counter. When we propagate, we cycle around the attributes which are responsible each modulo value of 3.
- Recomputing:
    - Then when update the value, we do different update based on the lazycounter.
        - If is a leaf node, we swap around the value stored in each attribute of the node that is responsible for different modulo value of 3.
            - However, here is where I was stuck on for the longest. For the recalculating step in the normal range tree, we recompute the sum from the lazy counter, then leave the lazy counter as it is. However, since we are recomputing the value based on the current value of lazy counter not the accumulated value of the lazy counter, that means we need to reset the lazy counter to 0 after every recalculation at the leaf node.
        - If it is not a leaf node, then we need to recompute the attribute value from the respective children's values

B – Problem set

- The first three operations are the normal ones that we've seen on lecture slide, simply max over range tree and sum over range tree. The difficult part are at the last 2 operations, which requires us to actually solve question within the tree

    - To find if the range is in increasing order, we need to have the following:
        - The children nodes responsible for subranges are both in increasing order
        - The left-most value of the right child is greater than the right-most value of the left child.
    - Same logic for decreasing order
- Hence, the information we need to include for every node to support all operations are:
    - Max value over the range
    - Sum over the range
    - Whether the range is in increasing order
    - Whether the range is in decreasing order
    - Left most value of the range
    - Right most value of the range
- Then for query, the most difficult part is also querying for increasing/decreasing order which got me stuck the longest for this question. We need to query by cases:
    - If the query requires querying both left and right child nodes, then we need to ensure both are in increasing/decreasing order, and the boundary also matches with the comparison.
    - If the query requires querying only one side, then we only need to ensure the side we queried is in increasing/decreasing order.

C – MEX queries:

- I did not solve this question, but I do have some thoughts on this question
- The values can go up to 10^18, which is too large to fit in a range tree, so we need to somehow compress the values
- I then noticed that not all the values in the range can be a possible solution. For any operation with range [l, r], only the values l, r, r+1 are possible candidates for solution. This is because for any missing value that has never been the l or r or r+1 of any query, that means the node on its left would have the same state (missing or present) as it, so it cannot be the smallest value. Also 1 is another possible candidate
- Following this logic, we can then run coordinate compression to reassign index to each solution candidates. We first put all the possible candidates into a map, then iterate through the keys of the map while keeping the counter which increase by 1 for every key we iterate thru, then let the value corresponding to the key equal to the counter.
- Then I don't know what to do afterwards for the operations.