A - Minimal string

- I was stuck on this question for quite a long time compare to B and C. One reason is that I was not so familiar with data structures when I just started this problem set, the other reason is that I found question B and C are quite similar to some questions I've done before, but not for A.
- The pattern I spotted is that to make string u lexicographically minimal, I want the smallest letter in s to be in u as early as possible. The range I search for this letter would be the last letter of t and the remaining letters of s. If I choose a letter from s, then I need to put the entire substring from the start of s till the minimal letter into t. I started working on this question before watching week2 lectures, so my initial approach was to use substring operations. My algorithm was working correctly, but it was too slow as I had to take substrings operations, and I stuck for a long time as I was tunnel visioning.
- After watching the lecture, I realized this can be done using a stack, as string t has a LIFO property. Hence my final solution is to traverse the string backwards, find the minimal character up to each letter. Then I push the first letter into the stack, and start a loop from the second letter.
    - If the top of the stack is the minimal letter, then I pop from the stack.
        - If the stack becomes empty, push the next character into the stack and increment index counter by 1
    - Otherwise, I push the next character in the string s into the stack and increment my index counter by 1

B – Classrooms

- I found this question very similar to the 3121 greedy algorithm question, where I had to allocate the most number of activities into one room, the only difference here is I have multiple rooms, but the logic remains the same. Hence the algorithm is to first order the activities by their finishing time, then keep allocating the earliest finishing activity, if it clashes with another activity, then allocate it to a different room. If there is no more room, then discard this activity.
- For the data structure to implement this algorithm, I found this algorithm quite similar to the Largest Interval question covered in the lecture, as both questions involve finding a value that is immediately greater than another value, which can be done using the "upper_bound" and "lower_bound" functions.
- Hence for the implementation, I make each activity into a pair using their starting and ending time, then sort by their ending time using a vector. Then I initialize a set which records the negative current ending time of each classroom. For each activity, I find the classroom that has the lates ending time that is before the starting time of this activity using upper_bound, since my set records negative value so this works. If I can find such classroom, then I pop that value from the set and insert ending time of the new activity into the set. Otherwise if I have spare room, then I push into the set, if not then discard this activity.

C - Xenia and Bit Operations

- This question is a straight forward implementation of range tree as I simply change the operation to XOR, and take the root of the tree as result.

D - Preparing for the Contest

- I started by noticing that if the best student is capable of fixing the most difficult bug, then it is guaranteed that this student can solve all the bugs. This “guarantee-ness” lead me to think about if binary search would be a viable strategy. I then started thinking what would be the monotonic behaviour exist in this question that allows me to use a binary search. My first thought is the sum of capabilities of students, but then realized a sequence of higher capability does not work wouldn’t guarantee a sequence of lower capability does not work, and vice versa, same for the sum of passes given.
- If neither capability nor passes exhibit monotonic behaviour, then the only restriction left is the number of days, and it does have a monotonic behaviour, as if the bugs can be fixed in C days, then it can also be fixed in C+k days for any integer k, vice versa. Hence the problem becomes how to determine if the bugs can be fixed in C days.
- The number of days k required to fix all bugs is determined by the students who solves the most number of bugs, that students needs to solve k bugs sequentially. In other words, all students can solve less than or equal to k bugs. The question now becomes a decision problem, where given the number of bugs every student can fix, find whether the minimum cost exceed the allowed number of passes given.
- My initial thought is to build a set of all students ordered by their capability. Then use lower bound to find the first student that can solve the current bug, then iterate to the end to find all the students that can solve the bug, then return the student of the lowest cost, and let that student to solve the next k bugs which equals to the number of days allowed. However, this requires me to create a set of all students from scratch at every run, then iterate through to find the min which is very inefficient.
- When I was discussing with Hugh about this question, he said “think about which data structure have you not used for this problem set.” Range tree and union find are definitely useless here, so the only option left would be heap/priority_queue. In the previous step, the main goal of the set is to find the students who are capable of solving a bug, then find the one with the minimum cost, this can be simply solved using a heap as I can push the cost of all the students into a min heap then pop. This student would then be responsible for the next k bugs, I then push the cost of all the students who are capable of solving the k+1th bug, then repeat. Terminating condition would be true if I solved all n bugs, or false if my heap is empty which means no one can solve this bug, or the passes given out exceed the allowance.
- To restore the sequence of student indexing, every time I pop a student from my heap, I would put the index of the student into an array of temporary solution buffer. If the function returns true, I will copy the content of temporary solution into an array of actual solution buffer, otherwise do nothing. The final solution would be in the actual solution buffer.