Task 1:

4a.

For the previous few cases that were wrong, I got the regularity condition wrong and most cases ended up falling into the not applicable part. The other cases such as case 1 and case 2 are very early and intuitive to solve therefore, I spent a bulk of my time trying to debug and look through the 5.6k test cases provided to try to find the differences. Ended up it was my regularity condition. Then after going through some examples and questions from the tutorial, I figured out that I missed the case where both d > 1 or a/b < c is good enough for the regularity check. Hence, my problem for this task was trying to figure out how to put the regularity condition into code. I also had a lot of nitty gritty printing errors that could be solved easily from printing the output into a file and examining them. I had problems dealing with the decimal cases and 1.0 cases then I found out that I made an error in my formatting function (it took in double instead of float). After changing it, the formatting became more accurate and correct.

4b.

From this, I learnt several shortcuts to calculate runtime faster using master theorem. So what I learnt was that d > 1 or a/b < c is enough to pass the regularity check and this is a very good shortcut to remember.

Master theorem pros are helping us calculate run time for conventional/normal recurrence equations, so we do not have to do them manually - pass the recurrence equations in and get a result fast. It can help us check our knowledge on master theorem as well.

Master theorem cons are that while it is good enough, it cannot return us the runtime for some of the recurrence equations. It does not cover several cases and sometimes we still have to get our hands dirty by solving those recurrence equation using other methods like recursion tree or telescoping method.

Task 2:

It is not likely that a comparison-based sorting algorithm can run under the time limit and pass the secret test case as the worst case time complexity for comparison-based algorithm is probably merge sort and in many cases if the merge sort is not implemented to perfection, it will fail the run time test. Therefore, this ensures that only linear time algorithm can be used as it have to be faster than nlogn for it to pass the time limit test. However, given that the host can perform 10^8 operations in 1 seconds and and the given time limit is 2.5s and maximum no. of TC is 77, it shows that the algorithm must have at most 192.5 * 10^8 operations. Anything more than that will exceed the time limit set by the question.

If all the numbers are non-negative and are in a similar range, we can always use counting sort or radix sort to optimise our run time complexity. These algorithm do not compare the integers.
I think that the numbers must be either non-negatives or all negatives (treat them as positive and get reverse array) and the numbers in the array must be in a similar range. Moreover, counting sort and radix sort will only work well with integers and not numbers with decimals points. We cannot use counting sort to sort decimal places. This extends to characters or other non-integer types as both using counting sort to keep count of the number of occurrences and throw them into their respective index. Another restriction might be very very large numbers as it might be hard for counting sort to create such a big array to hold these big numbers but we can always use radix sort and sort the numbers digits by digits.

My method uses counting sort. Initially, I've implemented radix sort that sorts 8-bit number at a time but I'm not too sure why time limit exceeded (surely I've gone wrong somewhere). So I decided to simplify it and just use counting sort instead. I let the maximum number be the maximum amount of bucket we keep for counting sort. Then, I go through the array to find numbers and throw them into the corresponding index (bucket) of the count array and afterwards retrieve them and put them back in order into the output array. I used a cumulative array as well to do it in a stable sorting manner (the way it was thought in lecture). Basically, I used a count array

to count the occurrences of each number and then created a cumulative count array and used that array to find out the position of each element such that the output is sorted. The time complexity for counting sort is O(n + k) as we are only iterating through the array once to throw them into their respective bucket. N is the length of the given array while k is the range of the inputs. K represents the range of numbers in the array. (To account for creating a count array of size K) Then again, retrieving them to put into an output array.

b)

The main difference between B1 and B2 is the size of the numbers. B1 maximum is 10^6 while B2 maximum is 10^9, and I think it is not possible to keep an array size of 10^9, so it is hard for us to do counting sort on it and since the number is so huge, it makes the time complexity very huge as counting sort time complexity is O(n+k). The upper bound now shifted to 10^9 so k = 10^9 potentially and this will increase the runtime by a lot. I've tried radix sort but still exceeded the time limit. I also thought about just initialising the array with indexes corresponding to the largest value but both failed to work. I tried using radix sort to sort 8-bit (tried till 16bit) of numbers at once but not too sure why I got time limit exceeded a few times then carry on to get run time error a few more times. I think it is because the count array that I am trying to create is abit too large and its causing problem in the memory section.