ChangingString:

* Initial thoughts: After reading the problems, the rules are very clear and easy to understand when there are examples provided as to shift a letter to its next or previous letter based on the number K passed in. But implementing it using codes feels like it’s on another level as I thought from the first place that I had to somehow convert string to integer, then calculate their total difference, and finally based on K, we need to subtract the difference or add 1 to total difference if no difference and then return the result.
* Reflections: After careful searching online, I found out that you could find the difference between letters immediately just by subtracting them directly, silly me. After this, I saved all of the diffs into a vector integer called diffs (using for-loop), sorted them in descending order in order to find out the biggest diff to subtract them later (or add if diffs = 0). After that, I set up a variable named total\_difference, calculating the total difference (to return final result). Finally, I iterated through vector diffs, adding 1 to total\_difference if encounter diffs[i] = 0, and subtracting by the number in diff[i] if encounter diffs[i] != 0 (in other word, return this diff to 0), and then return the final result, which is total\_difference.

MooingCows:

* Initial thoughts: The rules of the problem when first read might be tricky, but again, after reading the examples and compare them with provided algorithm to calculate the dissatisfaction, I could figure out how to calculate the total dissatisfaction of a cow to another cow mooing. To implement it into codes, first I thought about how to save the location of each cow on the string, then I set up a variable min of dissatisfaction to return the final result, but first I will store the most significant value to it in order to find the min dissatisfaction by comparing them (using for-loop first to iterate through the string to find other cow and calculate each dissatisfaction respectively).
* Reflections: I continued to expand the above solutions. First pass in vector type string into function, use n and m as number of rows and columns, set up vector pair int int to store cows’ location. Second, iterate through each row and column to find all cows and use emplace\_back to save cows’ location into vector cows. As minDissatisfaction is set up with highest possible integer, we continue to use for-loop to iterate through the number of cows. By using 2 different for-loop, one for the selected cow, the other for other remaining cows, we could calculate the dissatisfaction and then compare it to other calculated ones and return the minimum one (stored in minDissatifaction)

Reppity:

* Initial thoughts: The rules seem very clear for me when I first read it as to find the longest repeated string in a very long string. I got a bit confused when I started to read example 3 and 4 as I thought that string lowercase and string uppercase could still be counted if still the same letters, but no they aren’t. As to how to implement the rules by codes, first I know that the length of the repeated string is no longer than half the input, so I thought about looping the input to its half, then loop again inside that loop to extract each string from the first index 0 to the longest length possible for the repeated string, which is half the length of the input. Then I simply just find that extracted string inside the input to see if it’s in the input, then return the longest repeated string (integer).
* Reflections: As expanded from my first initial thought, I set up an interger maxLength in order to store the value and return it when finished. The outer loop, which is the end of extracted string (loop from half the input to the second index of input, and it is also the length of the extracted string) and the inner loop, which is the start of extracted string (loop from start of input to the length of input minus 2 times the respective length from the outer loop in order to iterate over all possible start positions for the substring and because the substring must appear at least twice, we need to leave enough space for the second occurrence), are used to find if that string is repeated in the second half of the input. Then if we found the extracted string inside the input, we save the length of that string into maxLength, and after the inner loop is finished and maxLength > 0, we break out of the outer loop since we found the longest repeated string. Finally, we return the result.

EggCartons:

* Initial thoughts: The rules of the problem is very easy to understand and also easier to code when compared to previous problems. The implementation of codes seems very straightforward as to implement a for-loops, which int eightEggs is smaller than n, and every iteration, it multiplies itself by 8, indicating 8 eggs per carton. After each loop, we get respective remaining eggs need to buy. And if that remaining can be divided by 6, then John can buy exactly n eggs and return the total cartons of eggs he needs to buy; else, return -1.
* Reflections: Expanded from initial thoughts: First of all, we need to set up our base variable minCarton to return, which is initially set by -1. Then we go through the loop. If John could buy exactly n eggs, we can set up a temporary variable named totalCartons to calculate the total cartons he needs to buy and set minCarton = totalCartons if minCarton = -1 (base condition) or totalCartons < minCarton from previous loop. And finally, we return the result.

BlackAndRed:

* Initial thoughts: The rules of the problem sounded very tricky when I first read it, even after reading all the examples. I thought first that still if we get red first and black later, we still won’t lose. But after careful reading and a while, I grasped the rules of the game as not to get more Red than Black at any time a card is revealed. As to how to implement the codes, I figured out that we need to get the size of the deck, then we use for-loop to iterate each position through the deck to cut. At each position, we cut the string from that index to the end, then we combine the rest of the cut deck (the deck from index to the end will be on the left of the deck). After we get the new deck, we check if the red card is not higher than black card when revealed using another loop for the deck. Finally, we return the min of the cut if found immediately (break immediately from the loop) as it is the min of all the possible cuts.
* Reflections: Beside from what I explained on my initial thoughts, there are some I want to add for easier reading. In the outer loop, beside red count and black count for card simulation, I also add boolean valid to check if the player won’t lose. Valid is true when red count < black count. If valid then the function loop will break immediately and return the cut\_pos, otherwise return -1. The new deck through each loop will always be renew from start to end of deck in order to find out the min possibility to cut the deck and help the player win. The inner loop will be for iterating through the new deck to simulate the game.