Code Jam 2018 - Qualification Round

Analysis: Saving The Universe Again

Test set 1

Since there is at most one `C` instruction in this test set, we can solve the two cases independently.

If there is no `C` instruction in **P**, then none of our swaps will have any effect, so all we can do is check whether the damage of the beam exceeds **D**.

If there is one `C` instruction in **P**, then we can try every possible position for the `C` instruction in the program. Assuming that there is at least one position for the `C` instruction that causes the total damage not to exceed **D**, we can choose the scenario that requires the fewest swaps; the number of required swaps for a scenario is equal to the distance between the original and final positions of the `C` instruction.

Test set 2

To solve test set 2, we will first derive a formula to compute the total damage based on the positions of the `C` and `S` instructions in **P**. Let N_C and N_S be the number of `C` and `S` instructions in **P**, respectively. Let C_i be the number of `S` instructions to the right of the i-th `C` instruction, where i uses 1-based indexing.

Note that the i-th `C` instruction will increase the damage of the subsequent beams by 2^{i-1}. For example, in the input program `CSSCSSCSS`, initially, all of the `S` instructions will inflict a damage of 1. Consider the damage dealt by the last `S` instruction. Since the robot has been charged twice, the damage output by the last instruction will be 4. Alternatively, we see that the damage, 4 = 1 (initial damage) + 2^0 (damage caused by the first `C`) + 2^1 (damage caused by the second `C`). By breaking down the damage by each `S` instruction in the same manner, the total damage output, D, of the input program is given by:

$$D = N_S + C_1 \times 1 + C_2 \times 2 + \ldots + C_{N_C} \times 2^{N_C - 1} .$$

Next, we investigate how each swap affects the amount of damage. A swap on adjacent characters which are the same will not affect the equation. When we swap the i-th `C` instruction with a `S` instruction to its right, the value of C_i will decrease by 1 since now there is one less S than before. On the other hand, swapping the i-th `C` instruction with an `S` instruction on its left will increase the value of C_i by 1. Note that in either case, we will only modify the value of C_i, and the other C values will remain the same. This suggests that we should only ever swap adjacent instructions of the form `CS`.

Therefore, executing M swaps is equivalent to reducing the values of C_is such that the total amount of reduction across all C_is is M. We want the total damage (according to the above equation) to be minimized. Clearly, we should reduce the values of C_i that contribute to the largest damage output, while making sure that each of the C_is is nonnegative.

Intuitively, all of this math boils down to a very simple algorithm! As long as there is an instance of `CS` in the current program, we always swap the latest (rightmost) instance. After each swap, we can recompute the damage and check whether it is still more than **D**. If it is not, then we can terminate the program. If we ever run out of instances of `CS` to swap, but the damage that the program will cause is still more than **D**, then the universe is doomed.