Editorial – elections

Authors: Andrei-Costin Constantinescu, Bogdan Ciobanu

Subtask 1 – O(N · Q), 28 points

Consider the following Greedy algorithm for each query L R:

- Initialize `balance` to `0`.
- Iterate from `1` to `N`.
- Whenever you encounter a 'C', increment `balance`.
- Whenever you encounter a 'T', decrement `balance`.
- If `balance` became negative (i.e. has value `-1`), nullify the vote that made `balance` go negative, and reset `balance` to `0`.
- Rerun the algorithm once more, this time going from `N` down to `1` and ignoring the already nullified votes.

To see why this algorithm is correct, we need to make a few observations:

1. Nullifying votes can only decrease the answer. Also, it can never turn a point in time with a non-negative `balance` into a point in time with a negative balance.
2. Within one pass, whenever `balance` becomes negative, at least one extra nullification is necessary, of one of the votes in the prefix / suffix which has already been considered.
3. After the first pass of the greedy algorithm, further nullifications done by the second pass can never invalidate what was done by the first pass (as a consequence of 2).
4. By doing the first pass nullifications as far to the right as possible, we ensure that the second pass will need the smallest possible number of additional nullifications (**Proof:** exchange argument – this way subtracting 1 from the `balance` during the second pass is delayed for as much as possible).

Subtasks 2-3 – O((N + Q) · log N), 100 points

For ease of writing, define `v[i] = 1` if the `i`-th character is a 'C', or `-1` otherwise.

We'll need an additional observation:

- The number of nullifications performed by the second pass of the greedy algorithm is given by `min(0, -minSuf)`, where `minSuf` is the minimum value `balance` reaches during the second pass of the greedy algorithm. It's not hard to see that this is also the minimum suffix sum of `v`.

With this in mind, reformulate the greedy algorithm as follows:

- Run the first pass of the usual greedy algorithm.
- Compute `minSuf` based on the suffix sums of `v`.

We'll process the queries in decreasing order of `L`. Iterate with a variable `i` from `N` down to `1`. Define `nullStk` as a stack of the positions which would be nullified by the modified greedy algorithm's first pass if run with `L = i`, `R = N`, ordered in decreasing order of index.

Updating `nullStk` turns out to be surprisingly easy:

- If `v[i] = -1`, then `i` gets pushed into the stack.
- If `v[i] = 1`, then we pop the stack, unless it's empty.

Exactly why this works is a matter of case analysis, and is left as an exercise to the reader.

To answer a query `L R`, one needs to:

- Binary search on `nullStk` in order to find out how many of the positions in `nullStk` do not exceed `R` (i.e. those positions which are going to be nullified by the first pass).
- Compute the value of `minSuf` on `v[L]`, `v[L + 1]`, …, `v[R]`, under the assumption that we updated `v[i]` to `0` whenever we pushed `i` into the stack and returned it to its original value whenever we popped it from the stack (i.e. compute the minimum suffix sum of `v[L]`, `v[L + 1]`, …, `v[R]`).

Computing `minSuf` on various ranges of `v` interleaved with value changes in `v` can be done in time O(log N) per operation using a segment tree. We leave most of the implementation details to the reader, but note that each node of the segment tree needs to store 4 integers:

1. `l` – the left bound of the interval
2. `r` – the right bound of the interval
3. $sum = \sum_{i=l}^{r} v[i]$
4. `minSuf` – the minimum suffix sum of `v[L]`, `v[L + 1]`, ..., `v[R]`, taking the empty suffix into account