香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

T26G2 - Closing Ceremony

Daniel Hsieh {QwertyPi}

2026-03-28

Background

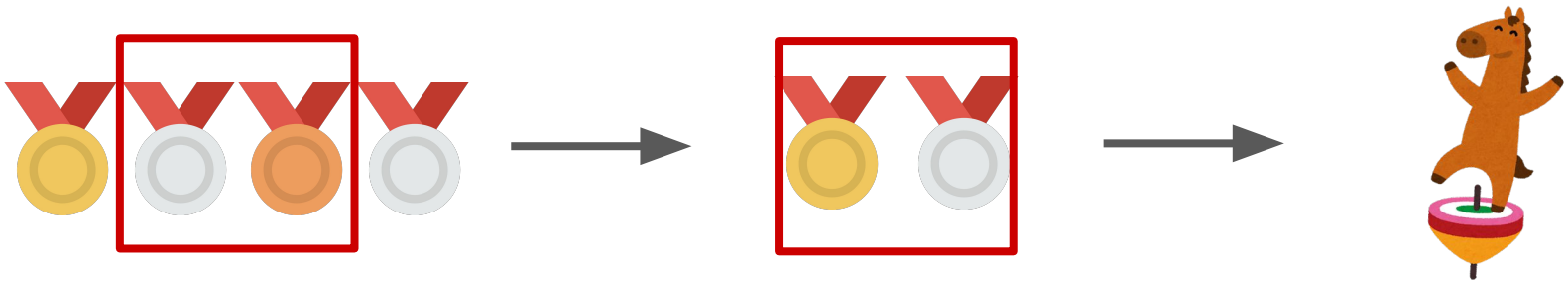
Problem Idea by `QwertyPi`

Preparation by `__jk__` and `happypotato`

Disclaimer: the background story is purely fictional, and any resemblance to real-life events is completely coincidental.

Problem Statement

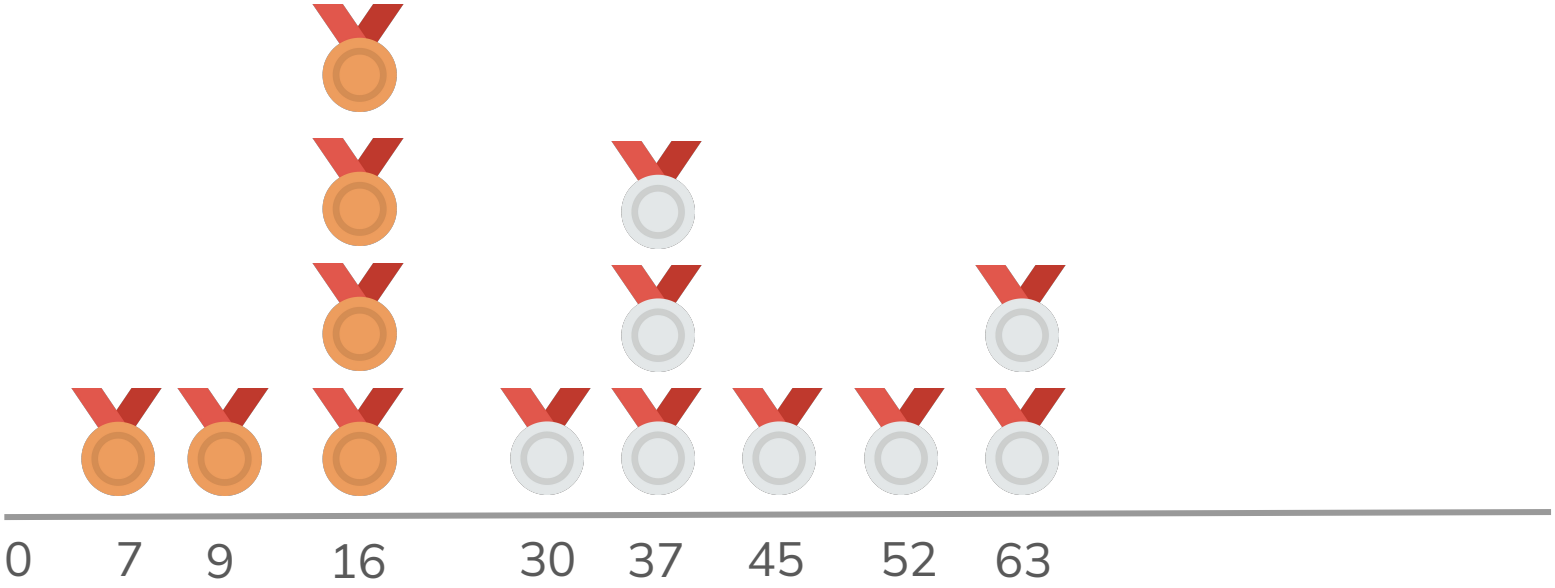
Given a string S of length `N` ($\leq 2 \times 10^5$), which consists of "`GSB`" only, count the number of substrings S[`l..r`] which we can remove all characters, by repeatedly **removing two adjacent characters that are different**.



Subtasks

Subtask	Score	N	Additional Constraints
1	7	$\leq 2 \times 10^5$	S = BSBS...BS
2	9	= 4	/
3	21	≤ 500	S contains B and S only
4	23	$\leq 2 \times 10^5$	
5	15	≤ 5000	/
6	11	$\leq 2 \times 10^5$	S does not contain BB or SS
7	14		/

Statistics



Subtask 1 (7%): S = BSBS...BS

Observation 1. If the length of a substring is **odd**, it can **never be vanishing**.

- Since two characters are removed in each operation, the length will always be odd, and never be zero.

Observation 2. A substring of `BS...BS` is **vanishing** iff **the length is even**.

- We can simply remove first two characters of the substring repeatedly.

Therefore, the answer is simply the number of even-length strings in `S`, which can be computed by `(N - 1) + (N - 3) + ... + 3 + 1 = N^2 / 4`.

Expected Score: 7 (Cumulative: 7)

Subtask 2 (9%): N = 4

How can we tell, in general, whether an **even** substring is vanishing or not?

Since `N` is small, you can either simply

- Try out all the ways to delete adjacent characters with recursion, or
- Find all vanishing strings of length at most 4, and simply check each substring one by one.

Expected Score: 9 (Cumulative: 16)

Subtask 3 (21%): N ≤ 500, S does not contain G

Let's take one step back — what if S only contains two characters, B and S?

Observation 3. Each operation must remove exactly one B and one S.

- If the number of B and S are different, it cannot be vanishing.

Observation 4. If a string contains both B and S, then we must be able to find either BS or SB as a substring.

Therefore, string S is vanishing iff the number of B and S are equal!

You can maintain the frequency arrays either naively or with partial sum.

Time Complexity: $O(N^3)$ or $O(N^2)$

Expected Score: 21 (Cumulative: 37)

Subtask 4 (23%): S does not contain G

How to speed up counting substrings with the same number of B and S?
By treating B as +1 and S as -1 and writing partial sum `ps[i]` := sum of first i values, we can count the number of pairs `(l, r)` such that `ps[l] = ps[r + 1]`.
This can be done by

```
for r in 0 .. N:
    ans += freq[ps[r]]
    freq[ps[r]] += 1
```

		B	B	S	B	S
		+1	+1	-1	+1	-1
ps	0	1	2	1	2	1

Time Complexity: O(N)
Expected Score: 44 (Cumulative: 60)

Subtask 5 (15%): N ≤ 5000

Define the number of occurrences of `G` in the substring S[`l`..`r`] as cnt_G, and similarly define cnt_S and cnt_B.

Claim: a substring S[`l`..`r`] is vanishing if and only if `r - l + 1` is even, and $\max(cnt_G, cnt_S, cnt_B) \le (r - l + 1) / 2$.

Proof:

- If `r - l + 1` were odd, then by Observation 1, it cannot be vanishing.
- Note that to make the substring S[`l`..`r`] become empty, we shall perform exactly `(r - l + 1) / 2` operations.

Subtask 5 (15%): N ≤ 5000

Claim: a substring S[l . . r] is vanishing if and only if r - l + 1 is even, and $\max(cnt_G, cnt_S, cnt_B) \le (r - l + 1) / 2$.

Proof:

- If any character has more than (r - l + 1) / 2 occurrences, then because each operation removes at most one occurrence of it, that character would still occur after (r - l + 1) / 2 operations.
  So, the substring cannot be vanishing.

Subtask 5 (15%): N ≤ 5000

Claim: a substring S[l . . r] is vanishing if and only if r - l + 1 is even, and $\max(cnt_G, cnt_S, cnt_B) \le (r - l + 1) / 2$.

But why does every substring with $\max(cnt_G, cnt_S, cnt_B) \le (r - l + 1) / 2$ work?

Proof:

- We induct on the length of the substring r - l + 1.
- Without loss of generality, we can assume that
  $cnt_B \le cnt_S \le cnt_G \le (r - l + 1) / 2$
  (otherwise, we can simply relabel the characters in the substring).

Subtask 5 (15%): N ≤ 5000

Claim: a substring S[l . . r] is vanishing if and only if r - l + 1 is even, and $\max(cnt_G, cnt_S, cnt_B) \le (r - l + 1) / 2$.

Proof:

- Assume $cnt_B \le cnt_S \le cnt_G \le (r - l + 1) / 2$.
  - If $cnt_S = cnt_G = (r - l + 1) / 2$, then pick any neighbouring pair of GS or SG
    - This reduces both cnt_S and cnt_G by 1, and also (r - l + 1) / 2 by 1 in one operation.
  - Otherwise, we pick an arbitrary substring of GS, SG, GB, or BG
    - This reduce cnt_G by 1, and also (r - l + 1) / 2 by 1 in one operation.
- So, we are always able to repeatedly do this, resulting in an empty string.

Subtask 5 (15%): N ≤ 5000

Claim: a substring S[l . . r] is vanishing if and only if r - l + 1 is even, and $\max(cnt_G, cnt_S, cnt_B) \le (r - l + 1) / 2$.

With the Claim above, we can exhaust l and iterate r in increasing order to check whether each substring S[l . . r] is vanishing or not by maintaining the frequencies of G, S, and B in the current substring.

Time Complexity: $O(N^2)$

Expected Score: 45 (Cumulative: 75)

Subtask 6 (11%): S does not contain BB or SS

What does the subtask condition imply?

- We always have cnt_S, $cnt_B \le ceil((r - l + 1) / 2)$ for any substring $S[l..r]$.

Since all vanishing substrings must have even length (by Observation 1), we can actually ignore the ceil and just treat it as $(r - l + 1) / 2$.

So, it remains to check whether cnt_G is $\le (r - l + 1) / 2$ or not.

How can we do this?

Subtask 6 (11%): S does not contain BB or SS

- Similar to Subtask 4 (where S contains `B` and `S` only), we can treat
  - each `G` to have value `+1`,
  - each `S` to have value `-1`,
  - each `B` to have value `-1`.

- Then, a substring S[`l..r`] is vanishing iff the sum of values in the substring is ≤ 0.

Nice, but how can we compute the answer efficiently?

Subtask 6 (11%): S does not contain BB or SS

In Subtask 4, we computed the partial sum array `ps[i]` := sum of first i values. We reuse the idea to do something similar here!

- This time, instead of counting the number of pairs `(l, r)` such that `ps[l] = ps[r + 1]`, we now count the pairs `(l, r)` that satisfy:
  - `ps[l] ≥ ps[r + 1]`, and
  - `r - l + 1` is even.

Subtask 6 (11%): S does not contain BB or SS

This can be conveniently maintained with any of your favourite data structures (std::multiset, BIT, segment tree...).

Time Complexity: O(N log N)

Expected Score: 18 (Cumulative: 86)

Subtask 7 (14%): No additional constraints

Observation 5. For any substring S[l . . r], at most one of the following holds:

- $cnt_G > (r - l + 1) / 2$
- $cnt_S > (r - l + 1) / 2$
- $cnt_B > (r - l + 1) / 2$.

Why? Because (r - l + 1) / 2 is half of the length of the substring S[l . . r], and it is obviously impossible to have two different characters, each occurring more than half the time in the substring!

Subtask 7 (14%): No additional constraints

Instead of counting how many vanishing substrings there are, we count how many substrings are ***not vanishing***.

- For each character c = `G`, `S`, or `B`, we can use the solution of Subtask 6 to find the number of vanishing substrings which have $cnt_c > (r - l + 1) / 2$.

The answer is then the total number of even-length substrings, minus the number of substrings which are ***not vanishing*** for each of the three cases.

Time Complexity: O(N log N)

Expected Score: 100

Remarks

- Always try to simplify your implementation! Even through the array given consists of `GSB`, doesn't mean you have to stick with this naming
- E.g. Instead of `cnt_G`, `cnt_S`, `cnt_B`, you can map `G` to `2`, `S` to `1` and `B` to `0`, and use a single `cnt[3][N + 1]` array to compute the same thing easier

- In general, if the most "trivial solution" is exponential in time, then your first target is to find any algorithm that runs in polynomial time, so that you can optimise upon it