Kick Start 2022 - Coding Practice with Kick Start Session #2

Analysis: Building Palindromes

View problem and solution walkthrough video

For each of Bob's questions, we have to figure out if it is possible to rearrange a string of letters to form a palindrome. The set of letters is the substring $[\mathbf{L_i}, \mathbf{R_i}]$ from the original string from the test case.

One possible approach is to enumerate all permutations of the substring and check if any are palindromes. For a substring of length l, this would involve checking up to $Factorial(l)$ permutations, each of which costs $O(l)$ to check if the permutation is a palindrome. Sometimes, "brute force" solutions like this are sufficient to solve competition problems, but in this case, even the $\mathbf{N}$-max of 20 from the smaller test set would take on the order of $20 \times Factorial(20) \approx 4.9 \times 10^{19}$ operations to perform. This is orders of magnitude more than can be accomplished within the time limit.

One important observation is that a palindrome can be described as an arbitrary string of zero or more letters, followed by the reverse of that string, optionally with one extra letter between them. For example, `ABCCBA` (without a middle letter) and `ABCXCBA` (with a middle letter) are both palindromes. These examples illustrate an important property of palindromes, which is that all letters, with the exception of the optional middle letter, must occur an even number of times.

This property holds true for all palindromes, and it turns out that a set of letters with this property can *always* be arranged to form a palindrome. We can do this by selecting pairs of letters and using them to construct two identical strings. Eventually, we will have at most one letter left over. Finally, we can take one of the strings and concatenate the reverse of the other, optionally with the leftover letter in between. The resulting string will be a palindrome!

Test set 1

For each question, we can count the frequencies of all letters in the substring and decide if it is possible to rearrange them to form a palindrome. For each question, the cost of calculating the frequency of each letter is $O(\mathbf{N})$. With $\mathbf{Q}$ questions per test case, this approach has a complexity of $O(\mathbf{Q} \times \mathbf{N})$, which is sufficient for Test set 1.

Test set 2

This test set has much larger limits for $\mathbf{Q}$ and $\mathbf{N}$, so the approach from Test set 1 will be too slow to complete within the time limit. An important observation is that for a given test case, Bob's questions are not independent - rather, they are all questions about substrings of the *same* string. Can we use this to our advantage and share or reuse calculations among the questions?

Thinking about the approach used in Test set 1, we can observe that the process of counting the letters in each substring ends up being repeated many times between the different questions. For example, if one question asked about the range $[20, 70]$ and another asked about the range $[30, 80]$, these both involve repeating the same counting process for the range $[30, 70]$. It turns out we can perform the counting process for *every possible question* ahead of time by generating a prefix sum for the frequency of each letter.

Generating a prefix sum for the letter frequencies in a string of length $\mathbf{N}$ has a complexity of $O(\mathbf{N})$. Furthermore, once the prefix sum is generated, answering any of Bob's questions can be done in constant time! For any question $[\mathbf{L_i}, \mathbf{R_i}]$, the frequency of letters in the substring is just the difference in value between the prefix sum at the beginning and the end of the substring. This reduces the computational complexity of each test case from $O(\mathbf{Q} \times \mathbf{N})$ to $O(\mathbf{Q} + \mathbf{N})$, which is sufficient for Test set 2.

It is also worth mentioning that while the 1GB memory limit is more than sufficient for this approach, we don't actually need to store the letter frequencies themselves. Because we only need to check whether a letter occurs an even or odd number of times, we can actually calculate a "prefix parity" and compare the parity at the beginning and end of a substring. If at most one letter has a different parity, then the substring can be arranged to form a palindrome.

Example Solution

```
Solve(N, Q, letters, questions) {
  answer = 0
  prefix[1]['A':'Z'] = 0
  for i in 1:N {
    prefix[i+1] = prefix[i]
    prefix[i+1][letters[i]]++
  }
  for i in 1:Q {
    [L,R] = questions[i]
    frequencies = prefix[R+1] - prefix[L]
    odds = 0
    for c in 'A':'Z' {
      if IsOdd(frequencies[c]) {
        odds++
      }
    }
    if odds <= 1 {
      answer++
    }
  }
  return answer
}
```