📓 Unique Substrings in Wraparound String — Documentation

1. Problem Statement

We define the wraparound string as an infinite repetition of "abcdefghijklmnopqrstuvwxyz" in a circular manner. For example:

"...zabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcd..."

Given a string s, determine how many unique non-empty substrings of s are present in this infinite wraparound string.

2. Intuition

To find valid substrings:

- Only consecutive substrings (like "abc", "zab", etc.) that follow the wraparound order are valid.
- Instead of generating all substrings (which is too slow), we track the maximum length of a valid substring ending with each character.
- If we know the longest valid substring ending with a character, we can count all shorter substrings from that sequence as well.

3. Key Observations

- Any substring that is not in consecutive alphabetical (or wraparound) order is not valid.
- For each character 'a' to 'z', if we track the maximum length of a valid substring ending at that character, then the number of unique substrings contributed by that character is equal to that length.
- Use modulo to handle the wraparound from 'z' to 'a'.

4. **Approach**

- Initialize a list of size 26 to track the maximum valid substring length ending with each character.
- Use a variable curr_len to track the length of the current valid sequence.
- Iterate through the string:
    - If the current character is the next in wraparound order of the previous one, increment curr_len.
    - Otherwise, reset curr_len to 1.
- Update the max_len_end_with[char] if curr_len is greater.
- Sum all values in max_len_end_with to get the total number of unique substrings.

5. **Edge Cases**

- Single character (e.g., "a") → Only one substring.
- Non-consecutive characters (e.g., "cac") → Count individual valid substrings only.
- Wraparound sequences (e.g., "zab") → Should be handled using modulo logic.
- Repeated characters → Ensure substrings are counted uniquely, not by total occurrence.

6. **Complexity Analysis**

🕒 Time Complexity: O(n)

- We traverse the string once.
- Each character is processed in constant time.

💾 Space Complexity: O(1)

- Only 26 elements stored in max_len_end_with.

7. Alternative Approaches

Method	Description	Time	Space
Brute-force	Generate all substrings and check if valid	$O(n^2)$ to $O(n^3)$	High
Optimized (current)	Track longest substring per character	$O(n)$	$O(1)$

Brute-force fails on large inputs due to time limits, so the optimized solution is preferred.

8. Test Cases

Input	Output	Explanation
"a"	1	Only one substring "a"
"cac"	2	Valid substrings: "a", "c"
"zab"	6	"z", "a", "b", "za", "ab", "zab"
"abcabc"	6	Maximum unique substrings are only up to "abc"

9. Final Thoughts

- This problem is a great example of how pattern recognition and mathematical reasoning can replace brute-force solutions.
- By leveraging the structure of the alphabet and the idea of tracking the maximum valid lengths, we reduce the complexity from $O(n^2)$ to $O(n)$.
- It's an efficient and elegant solution to what could be a very costly problem computationally.