**[1400. Construct K Palindrome Strings](https://leetcode.com/problems/construct-k-palindrome-strings/)**

Given a string s and an integer k, return true *if you can use all the characters in*s*to construct*k*palindrome strings or*false*otherwise*.

**Example 1:**

**Input:** s = "annabelle", k = 2

**Output:** true

**Explanation:** You can construct two palindromes using all characters in s.

Some possible constructions "anna" + "elble", "anbna" + "elle", "anellena" + "b"

**Example 2:**

**Input:** s = "leetcode", k = 3

**Output:** false

**Explanation:** It is impossible to construct 3 palindromes using all the characters of s.

**Example 3:**

**Input:** s = "true", k = 4

**Output:** true

**Explanation:** The only possible solution is to put each character in a separate string.

**Constraints:**

* 1 <= s.length <= 105
* s consists of lowercase English letters.
* 1 <= k <= 105

1400. Construct K Palindrome Strings Given a string s and an integer k, return true if you can use all the characters in s to construct k palindrome strings or false otherwise. Example 1: Input: s = "annabelle", k = 2 Output: true Explanation: You can construct two palindromes using all characters in s. Some possible constructions "anna" + "elble", "anbna" + "elle", "anellena" + "b" Example 2: Input: s = "leetcode", k = 3 Output: false Explanation: It is impossible to construct 3 palindromes using all the characters of s. Example 3: Input: s = "true", k = 4 Output: true Explanation: The only possible solution is to put each character in a separate string. Constraints: 1 <= s.length <= 105 s consists of lowercase English letters. 1 <= k <= 105

To determine if it's possible to construct kkk palindrome strings using all the characters in the string sss, we can break the problem into logical steps:

**Key Observations:**

1. **Character Frequency and Palindromes**:
   * A palindrome string has at most one character with an odd frequency (in the case of odd-length palindromes).
   * For a set of kkk palindromes, you can assign characters with odd frequencies to different palindromes. This means that the number of palindromes cannot be less than the count of characters with odd frequencies.
2. **Minimum and Maximum Constraints**:
   * The minimum number of palindromes kkk must be at least equal to the count of odd frequency characters in sss.
   * The maximum number of palindromes kkk is bounded by the total length of sss, since each palindrome must have at least one character.

**Approach:**

1. Count the frequency of each character in sss.
2. Count how many characters have odd frequencies.
3. Check if kkk is within the valid range:
   * odd\_count≤k≤len(s)\text{odd\\_count} \leq k \leq \text{len}(s)odd\_count≤k≤len(s).

**Algorithm:**

1. Use a frequency counter to calculate the count of each character.
2. Count the number of odd frequency characters.
3. Return true if kkk satisfies the constraints, otherwise return false.

**Implementation:**

Here's the Python solution:

python

Copy code

def canConstruct(s: str, k: int) -> bool:

# Step 1: Count character frequencies

from collections import Counter

freq = Counter(s)

# Step 2: Count odd frequency characters

odd\_count = sum(1 for count in freq.values() if count % 2 != 0)

# Step 3: Check if k is within the valid range

return odd\_count <= k <= len(s)

# Example Usage

print(canConstruct("annabelle", 2)) # Output: True

print(canConstruct("leetcode", 3)) # Output: False

print(canConstruct("true", 4)) # Output: True

**Explanation of the Code:**

1. **Character Frequency**:
   * We use collections.Counter to efficiently calculate the frequency of each character in sss.
2. **Odd Count**:
   * For each character's count, we check if it is odd. The sum of all odd counts gives the minimum number of palindromes needed.
3. **Validation**:
   * Check if kkk satisfies the range constraint (odd\_count≤k≤len(s)\text{odd\\_count} \leq k \leq \text{len}(s)odd\_count≤k≤len(s)).
   * If it does, return True; otherwise, return False.

**Complexity:**

* **Time Complexity**: O(n)O(n)O(n), where nnn is the length of sss (to count frequencies).
* **Space Complexity**: O(26)=O(1)O(26) = O(1)O(26)=O(1), as the input consists of only lowercase English letters.

Runtime Error NameError: name 'Solution' is not defined ^^^^^^^^ ret = Solution().canConstruct(param\_1, param\_2) Line 46 in \_driver (Solution.py) \_driver() Line 61 in <module> (Solution.py)

The error indicates that the code expects a Solution class with a method named canConstruct. To match the expected structure, we need to define the Solution class and place the canConstruct method inside it. Here's how the code should be structured:

**Corrected Code:**

python

Copy code

class Solution:

def canConstruct(self, s: str, k: int) -> bool:

# Step 1: Count character frequencies

from collections import Counter

freq = Counter(s)

# Step 2: Count odd frequency characters

odd\_count = sum(1 for count in freq.values() if count % 2 != 0)

# Step 3: Check if k is within the valid range

return odd\_count <= k <= len(s)

# Example Usage

# Create an instance of Solution and call the canConstruct method

solution = Solution()

print(solution.canConstruct("annabelle", 2)) # Output: True

print(solution.canConstruct("leetcode", 3)) # Output: False

print(solution.canConstruct("true", 4)) # Output: True

**Explanation of Fix:**

1. **Class Definition**:
   * The code expects a Solution class. This is a common requirement in competitive programming platforms or online coding tests (e.g., LeetCode, HackerRank).
2. **Method Placement**:
   * The canConstruct function is placed as a method inside the Solution class.
3. **Method Invocation**:
   * Create an instance of the Solution class and call the canConstruct method on it.

**Testing:**

This structure will work on platforms requiring a Solution class. If you're still encountering errors, verify that you're using the correct driver code provided by the platform.

from collections import Counter class Solution: def canConstruct(self, string: str, num\_partitions: int) -> bool: # If the length of the string is less than the required number of partitions, # we cannot construct the required partitions, so return False. if len(string) < num\_partitions: return False # Count the frequency of each character in the string using Counter. char\_counter = Counter(string) # Calculate the number of characters that have an odd count. # This loop goes through the values (counts) in the char\_counter # and uses a bitwise AND operation (&) with 1 to determine if the count is odd. odd\_count = sum(count & 1 for count in char\_counter.values()) # The number of characters with odd counts should not exceed the number # of partitions we want to create, because each palindrome must have at most # one character with an odd count (in the middle). Therefore, if we have # no more odd counts than the number of partitions, we can construct the palindromes. return odd\_count <= num\_partitions

The provided solution is clean and correctly addresses the problem using efficient logic.

Here's a step-by-step breakdown of what the code does:

**Step-by-Step Explanation:**

1. **Edge Case Check**:
   * If the length of the string (len(string)\text{len(string)}len(string)) is less than the required number of partitions (num\_partitions\text{num\\_partitions}num\_partitions), return False.
   * This is because we cannot split the string into more partitions than there are characters.
2. **Character Frequency Count**:
   * Use Python's collections.Counter to calculate the frequency of each character in the string efficiently.
3. **Odd Frequency Count**:
   * For a palindrome, at most one character in each partition can have an odd frequency.
   * Count the characters with odd frequencies using a sum over the values in the frequency dictionary. The condition count & 1 checks if the count is odd.
4. **Validation**:
   * If the number of odd frequency characters (odd\_count\text{odd\\_count}odd\_count) is less than or equal to num\_partitions\text{num\\_partitions}num\_partitions, it is possible to form the required number of palindrome partitions. Return True; otherwise, return False.

**Example Usage:**

python

Copy code

solution = Solution()

# Test cases

print(solution.canConstruct("annabelle", 2)) # Output: True

print(solution.canConstruct("leetcode", 3)) # Output: False

print(solution.canConstruct("true", 4)) # Output: True

print(solution.canConstruct("aaaa", 1)) # Output: True

print(solution.canConstruct("aabbcc", 3)) # Output: True

**Complexity:**

1. **Time Complexity**: O(n)O(n)O(n), where nnn is the length of the string. This is due to the single pass required to compute character frequencies and sum up odd counts.
2. **Space Complexity**: O(1)O(1)O(1) (or O(26)O(26)O(26)), as the Counter stores at most 26 entries (for lowercase English letters).

This solution is both efficient and concise, making it ideal for handling constraints like 1≤len(s),k≤1051 \leq \text{len}(s), k \leq 10^51≤len(s),k≤105.