**Documentation on Reconstruct Original Digits from English**

**Table of Contents**



## 1. Problem Statement

Given a string s containing an unordered English representation of digits (0-9), reconstruct and return the digits in ascending order.

**Example 1:**

**Input:**

s = "owoztneoer"

**Output:**

"012"

**Example 2:**

**Input:**

s = "fviefuro"

**Output:**

"45"

**Constraints:**

- $1 \le s.length \le 10^5$
- s[i] is one of the letters: ["e", "g", "f", "i", "h", "o", "n", "s", "r", "u", "t", "w", "v", "x", "z"]
- The input is guaranteed to be a valid jumbled representation of digits.

## 2. Intuition

The key to solving this problem efficiently is recognizing **unique character associations** for certain digits. Instead of trying all possible permutations, we leverage these unique character occurrences to determine the counts of each digit.

## 3. Key Observations

Each digit in English has specific characters that can be uniquely identified:

| Digit | Word | Unique Character |
|---|---|---|
| 0 | zero | **z** |
| 2 | two | **w** |
| 4 | four | **u** |
| 6 | six | **x** |
| 8 | eight | **G** |

After determining the above digits, we can find the remaining ones:

| Digit | Word | Dependent Characters |
|---|---|---|
| 1 | one | **o** (appears in zero, two, four) |
| 3 | three | **h** (appears in three, eight) |
| 5 | five | **f** (appears in five, four) |
| 7 | seven | **s** (appears in seven, six) |
| 9 | nine | **i** (appears in nine, five, six, eight) |

### 4. Approach

I. **Count the frequency of each character** in the input string.

II. **Identify numbers with unique characters** and record their frequency:
   a. 'z' → 0
   b. 'w' → 2
   c. 'u' → 4
   d. 'x' → 6
   e. 'g' → 8

III. **Deduct counts for remaining numbers** based on known values:
   a. 'o' → 1 (subtract counts of 0, 2, and 4)
   b. 'h' → 3 (subtract count of 8)
   c. 'f' → 5 (subtract count of 4)
   d. 's' → 7 (subtract count of 6)
   e. 'i' → 9 (subtract counts of 5, 6, and 8)

IV. **Reconstruct the output string** by arranging digits in sorted order.

### 5. Edge Cases

- **All digits present**: "zeroonetwothreefourfivesixseveneightnine"
- **Only one digit**: "nine" → Output "9"
- **Randomly shuffled characters**: "xwutgieorhzefoevns"
- **Large input**: Length close to 10510^5

**6. Complexity Analysis**

**Time Complexity**

- **Counting characters**: O(N)O(N)
- **Identifying digits**: O(1)O(1) (fixed 10 numbers)
- **Constructing the output**: O(1)O(1)
  **Total Complexity**: **O(N)O(N)**

**Space Complexity**

- We use a dictionary for character frequency (O(1)O(1)) and a dictionary for digit frequency (O(1)O(1)).
- The final result string takes at most O(N)O(N) space.
  **Total Complexity**: **O(N)O(N)**

**7. Alternative Approaches**

**Brute Force Approach**

1. Generate all permutations of the string.
2. Check if the permutation forms valid English digit words.
3. Sort and return the numbers.
   **Time Complexity: O(N!)O(N!) → Not feasible**

**Sorting by Word Length**

- Sort known words by length and extract numbers accordingly.
- **Not optimal** due to complex substring matching.

**8. Test Cases**

solution = Solution()

# Test Case 1

assert solution.originalDigits("owoztneoer") == "012"

```
# Test Case 2
assert solution.originalDigits("fviefuro") == "45"

# Test Case 3
assert solution.originalDigits("zeroonetwothreefourfivesixseveneightnine") == "0123456789"

# Test Case 4 (Random shuffle)
assert solution.originalDigits("xwutgieorhzefoevns") == "0245678"

# Test Case 5 (Single digit cases)
assert solution.originalDigits("nine") == "9"
assert solution.originalDigits("eight") == "8"
assert solution.originalDigits("two") == "2"

print("All test cases passed!")
```

## 9. Final Thoughts

- **Efficient approach** using **character frequency**.
- **Time Complexity O(N)O(N) is optimal** for large inputs.
- **Alternative approaches** exist but are significantly less efficient.
- **Edge cases are handled properly**, ensuring correctness.