# 423. Reconstruct Original Digits from English

Medium Hash Table Math String Leetcode Link

## Problem Description

The problem presents us with a string `s` that comprises jumbled letters representing English words for digits zero to nine. For example, the word for zero "zero" includes the letters "z", "e", "r", and "o". Our task is to rearrange these jumbled letters to figure out what digits they represent and then return those digits sorted in ascending order. As an example, if the input string is "owoztneoer", the output should be "012" corresponding to the digits one, zero, and two represented by the words "one", "zero", and "two" that can be formed from the given letters.

## Intuition

The solution approach is based on the observation that certain letters uniquely identify some numbers. For instance, the letter "z" only appears in "zero", and the letter "w" only shows up in "two". We can use these unique letters to immediately find how many times a particular digit occurs in the string.

The Python code implements this approach by creating a counter `cnt` of ten elements representing the digits 0 to 9. Then we fill in this counter based on the frequency of the unique identifiers:

- `z` appears only in "zero", so `cnt[0]` is set to the count of 'z'.
- `w` appears only in "two", so `cnt[2]` is set to the count of 'w'.
- `u` appears only in "four", so `cnt[4]` is set to the count of 'u'.
- `x` appears only in "six", so `cnt[6]` is set to the count of 'x'.
- `g` appears only in "eight", so `cnt[8]` is set to the count of 'g'.

After identifying and counting the digits with unique letters, the code then uses this information to determine the counts of other digits, which have letters that can be shared with the digits already counted. This is performed through subtraction:

- `h` appears in "three" and "eight", but since we've already counted the 'h' from "eight", `cnt[3]` can be calculated by subtracting `cnt[8]` from the count of 'h'.
- `f` appears in "five" and "four", but `cnt[4]` is already known, so subtract that from the count of 'f' to find `cnt[5]`.
- `s` appears in "seven" and "six", subtract `cnt[6]` from the count of 's' to get `cnt[7]`.
- For `cnt[1]`, it requires the count of the letter 'o', which appears in "zero", "one", "two", and "four". We've already found the counts for "zero", "two", and "four", so we subtract those from the count of 'o'.
- The digit 'nine' is represented by 'i', which appears in "five", "six", "eight", and "nine". Again, subtract the counts of "five", "six", and "eight", that have already been accounted for, from the count of 'i', to arrive at `cnt[9]`.

Finally, the code concatenates the digits together into a string, with each digit repeated as many times as it was counted, and returns this string as the result.

## Solution Approach

The solution utilizes several powerful concepts in Python, such as the `Counter` class from the `collections` module for frequency counting and list comprehension for building the final output.

Here's a step-by-step walkthrough:

1. First, the program imports the `Counter` class from the `collections` module, which is used to count the frequency of each character in the input string `s`.

2. `counter = Counter(s)`: This line creates an instance of the `Counter` object that contains the frequency of each character in the string.

3. The `cnt` list of length 10 is initialized to store the count of each digit from 0 to 9. Initially, all values in `cnt` are set to 0.

4. The code then identifies the counts of digits with unique characters:

   - `cnt[0]` is set to the count of `'z'`, which only appears in "zero".
   - `cnt[2]` is set to the count of `'w'`, which only appears in "two".
   - And so on for `cnt[4]`, `cnt[6]`, and `cnt[8]`.
5. With those unique digit counts determined, the counts for the other digits that share characters can now be calculated:

   - `cnt[3]` is the count of `'h'` (from "three" and "eight") minus `cnt[8]`.
   - `cnt[5]` is the count of `'f'` (from "five" and "four") minus `cnt[4]`.
   - `cnt[7]` is the count of `'s'` (from "seven" and "six") minus `cnt[6]`.
   - `cnt[1]` is the count of `'o'` (from "zero", "one", "two", "four") minus the counts of `cnt[0]`, `cnt[2]`, and `cnt[4]`.
   - `cnt[9]` is the count of `'i'` (from "five", "six", "eight", "nine") minus `cnt[5]`, `cnt[6]`, and `cnt[8]`.
6. Finally, once all the counts are determined, the code uses list comprehension to construct the output string. This is done with `''.join(cnt[i] * str(i) for i in range(10))`, which iterates over the digits 0 through 9, repeating each digit a number of times equal to its count, and then joining them together into one string.

This code is a clever exploit of the uniqueness of certain characters in the English language spelling of these digit words and the smart use of character frequency counting to compute the overall digit counts indirectly.

### Example Walkthrough

Let's illustrate the solution approach using a small example. Suppose our input string is `s = "nnei"` which we need to organize into sorted digits.

1. We start by using the Counter class to determine the frequency of each character:

```python
from collections import Counter
counter = Counter(s)  # s = 'nnei'
```

After this step, `counter` looks like this: `Counter({'n': 2, 'e': 1, 'i': 1})`.

2. We initialize an array `cnt` with 10 zeroes to hold the counts for each digit:

```python
cnt = [0] * 10
```

3. Next, we can deduce the unique counts for some digits; however, in our example `s = "nnei"`, there are no unique digit identifiers like "z" for zero, "w" for two, "u" for four, "x" for six, or "g" for eight. Thus, values in `cnt` remain unaltered for these particular digits.

4. From the remaining characters, we infer the following:

   - The letter 'n' appears twice, and since it can only be part of "nine" or "one", we don't have enough information to determine the count directly.
   - The letter 'e' appears once, which is shared by "one", "three", "five", "seven", and "nine".
   - The letter 'i' appears once; it can be from "five", "six", "eight", or "nine".
5. Based on our Counter, we can take an educated guess. The only digit that can be formed is "nine", which uses all the letters in our counter: 'n', 'i', 'n', 'e'. Therefore, we set `cnt[9]` to 1:

```python
cnt[9] = counter['i']  # counter['i'] is 1 for 'nine'
```

After this, it'll also be clear that since we have identified 'nine' and have used up all 'n', 'e' and 'i' from the string, `cnt[1]`, `cnt[3]`, `cnt[5]`, and `cnt[7]` will not be incremented as there are no remaining counts of 'n', 'e', or 'i'.

6. Following these deductions, we can now construct our result. Since only 'nine' can be formed, our result string will only contain '9':

```python
result = ''.join(str(i) * cnt[i] for i in range(10))  # generates '9'
```

After these steps, the output string is '9', which is the expected result for the input string `s = "nnei"`.

## Python Solution

```python
from collections import Counter

class Solution:
    def original_digits(self, s: str) -> str:
        # Create a counter object to count occurrences of each character in the string
        char_count = Counter(s)
        # Initialize a list to store the count of each digit from 0 to 9
        digit_count = [0] * 10

        # Each unique character can identify a number:
        # 'z' is present only in "zero" \
        digit_count[0] = char_count['z']
        # 'w' is present only in "two"
        digit_count[2] = char_count['w']
        # 'u' is present only in "four"
        digit_count[4] = char_count['u']
        # 'x' is present only in "six"
        digit_count[6] = char_count['x']
        # 'g' is present only in "eight"
        digit_count[8] = char_count['g']

        # Now we proceed to find the count of the other numbers
        # 'h' is present in "three" and "eight", but we have already counted "eight" occurrences
        digit_count[3] = char_count['h'] - digit_count[8]
        # 'f' is present in "five" and "four", but we have already counted "four" occurrences
        digit_count[5] = char_count['f'] - digit_count[4]
        # 's' is present in "seven" and "six", but we have already counted "six" occurrences
        digit_count[7] = char_count['s'] - digit_count[6]

        # For ones, it is present in "one", "zero", "two", and "four", which are already counted
        digit_count[1] = char_count['o'] - digit_count[0] - digit_count[2] - digit_count[4]
        # 'i' is present in "nine", "five", "six", and "eight", which are already counted
        digit_count[9] = char_count['i'] - digit_count[5] - digit_count[6] - digit_count[8]

        # Construct the final string with the counted digits
        result = ''.join(str(i) * digit_count[i] for i in range(10))
        return result

```

## Java Solution

```java
class Solution {
    public String originalDigits(String s) {
        // Create an array to keep count of all alphabet characters within the string
        int[] charCounter = new int[26];
        // Count the frequency of each character in the string
        for (char c : s.toCharArray()) {
            charCounter[c - 'a']++;
        }

        // Create an array to count the occurrences of each digit (0-9) in the string
        int[] digitCounts = new int[10];

        // Count unique letters that only appear in a single number's spelling.
        // This gives us a definitive count of certain digits.
        digitCounts[0] = charCounter['z' - 'a']; // Zero
        digitCounts[2] = charCounter['w' - 'a']; // Two
        digitCounts[4] = charCounter['u' - 'a']; // Four
        digitCounts[6] = charCounter['x' - 'a']; // Six
        digitCounts[8] = charCounter['g' - 'a']; // Eight

        // For other numbers which share letters, we subtract the counts of
        // already identified unique ones to get the correct digit counts.
        digitCounts[3] = charCounter['h' - 'a'] - digitCounts[8];  // Three (h is also in eight)
        digitCounts[5] = charCounter['f' - 'a'] - digitCounts[4];  // Five (f is also in four)
        digitCounts[7] = charCounter['s' - 'a'] - digitCounts[6];  // Seven (s is also in six)

        // For one and nine, we deduce their counts by subtracting the counts of digits
        // that share the same letters and have been previously determined.
        digitCounts[1] = charCounter['o' - 'a'] - digitCounts[0] - digitCounts[2] - digitCounts[4]; // One
        digitCounts[9] = charCounter['i' - 'a'] - digitCounts[5] - digitCounts[6] - digitCounts[8]; // Nine

        // Construct the final digits string in ascending order.
        StringBuilder result = new StringBuilder();
        for (int i = 0; i < 10; ++i) {
            for (int j = 0; j < digitCounts[i]; ++j) {
                result.append(i);
            }
        }

        // Return the constructed string which represents the original digits in ascending order.
        return result.toString();
    }
}

```

## C++ Solution

```cpp
#include <vector>
#include <string>

class Solution {
public:
    string originalDigits(string s) {
        // Creating a frequency counter for each alphabet letter
        vector<int> alphaCounter(26, 0);
        for (char c : s) {
            ++alphaCounter[c - 'a'];
        }

        // Creating a frequency counter for digits 0 through 9
        vector<int> digitCounter(10, 0);

        // Counting occurrences of letters that uniquely identify a digit
        digitCounter[0] = alphaCounter['z' - 'a']; // The presence of 'z' indicates 'zero'
        digitCounter[2] = alphaCounter['w' - 'a']; // The presence of 'w' indicates 'two'
        digitCounter[4] = alphaCounter['u' - 'a']; // The presence of 'u' indicates 'four'
        digitCounter[6] = alphaCounter['x' - 'a']; // The presence of 'x' indicates 'six'
        digitCounter[8] = alphaCounter['g' - 'a']; // The presence of 'g' indicates 'eight'

        // Subtracting counts of digits where letters are shared with the unique-letters digits
        digitCounter[3] = alphaCounter['h' - 'a'] - digitCounter[8]; // 'three' is indicated by 'h', but 'eight' also contains 'h'
        digitCounter[5] = alphaCounter['f' - 'a'] - digitCounter[4]; // 'five' is indicated by 'f', but 'four' also contains 'f'
        digitCounter[7] = alphaCounter['s' - 'a'] - digitCounter[6]; // 'seven' is indicated by 's', but 'six' also contains 's'

        // For digits 'one' and 'nine', which share letters with multiple digits
        digitCounter[1] = alphaCounter['o' - 'a'] - digitCounter[0] - digitCounter[2] - digitCounter[4];
        digitCounter[9] = alphaCounter['i' - 'a'] - digitCounter[5] - digitCounter[6] - digitCounter[8];

        // Constructing the output string based on the digits' frequency
        string result;
        for (int i = 0; i < 10; ++i) {
            for (int j = 0; j < digitCounter[i]; ++j) {
                result += char(i + '0'); // Append the numeric digit i, converted to a character
            }
        }

        return result;
    }
};

```

## Typescript Solution

```typescript
// Define function to get the original digits from the input string.
function originalDigits(s: string): string {
    // Create a frequency counter for each alphabet letter
    const alphaCounter: number[] = new Array(26).fill(0);
    for (const char of s) {
        alphaCounter[char.charCodeAt(0) - 'a'.charCodeAt(0)]++;
    }

    // Create a frequency counter for digits 0 through 9
    const digitCounter: number[] = new Array(10).fill(0);

    // Count occurrences of letters that uniquely identify a digit
    digitCounter[0] = alphaCounter['z'.charCodeAt(0) - 'a'.charCodeAt(0)]; // 0 is signified by 'z'
    digitCounter[2] = alphaCounter['w'.charCodeAt(0) - 'a'.charCodeAt(0)]; // 2 is signified by 'w'
    digitCounter[4] = alphaCounter['u'.charCodeAt(0) - 'a'.charCodeAt(0)]; // 4 is signified by 'u'
    digitCounter[6] = alphaCounter['x'.charCodeAt(0) - 'a'.charCodeAt(0)]; // 6 is signified by 'x'
    digitCounter[8] = alphaCounter['g'.charCodeAt(0) - 'a'.charCodeAt(0)]; // 8 is signified by 'g'

    // Deduct counts of letters from previous findings to find the rest of the digits
    digitCounter[3] = alphaCounter['h'.charCodeAt(0) - 'a'.charCodeAt(0)] - digitCounter[8]; // 3 contains 'h', adjusted for '8'
    digitCounter[5] = alphaCounter['f'.charCodeAt(0) - 'a'.charCodeAt(0)] - digitCounter[4]; // 5 contains 'f', adjusted for '4'
    digitCounter[7] = alphaCounter['s'.charCodeAt(0) - 'a'.charCodeAt(0)] - digitCounter[6]; // 7 contains 's', adjusted for '6'

    // Special cases for 'one' and 'nine'
    digitCounter[1] = alphaCounter['o'.charCodeAt(0) - 'a'.charCodeAt(0)] - digitCounter[0] - digitCounter[2] - digitCounter[4]; //
    digitCounter[9] = alphaCounter['i'.charCodeAt(0) - 'a'.charCodeAt(0)] - digitCounter[5] - digitCounter[6] - digitCounter[8]; //

    // Construct the output string based on the digits' frequency
    let result: string = '';
    for (let i = 0; i < 10; ++i) {
        for (let j = 0; j < digitCounter[i]; ++j) {
            result += i.toString(); // Append the actual numeric digit i
        }
    }

    return result;
}

// Example of how to use the function
const inputString = "owoztneoer";
const outputDigits = originalDigits(inputString);
console.log(outputDigits); // Should print the sorted original digits found in the string

```

## Time and Space Complexity

The given Python code uses a `Counter` to track the frequency of characters in the string `s`, and then deduces the counts for each digit from 0 to 9. The final string is composed based on these counts.

### Time Complexity:

- Constructing the `Counter` takes `O(n)` time, where `n` is the length of the string `s`.
- The code includes ten direct assignments (for `cnt[0]` to `cnt[8]`) and three subtraction operations (for `cnt[9]`, `cnt[7]`, and `cnt[5]`) which all happen in constant time, i.e., `O(1)` each.
- The last operation consists of iterating over the range 0 to 9 and concatenating the digits the number of times they appear. This is `O(m)` where `m` denotes the total number of digits in the output. Note that `m` depends on the input `s` and its distribution of characters representing digits.

Adding all these operations together, we arrive at a final time complexity of `O(n + m)`. In the worst case, the output string `m` could be proportional to the input string `n`, which means `m` could be as big as `O(n)`. Therefore, the overall time complexity simplifies to `O(n)`.

### Space Complexity:

- The `Counter` which stores the frequency of each character needs space up to `O(k)`, where `k` is the number of unique characters in string `s`. Since we are dealing with English letters and a few additional characters, `k` is bounded by a constant (at most 26 for letters plus additional characters for digits), so we can consider it `O(1)`.
- The `cnt` array uses constant space because it is of a fixed size of 10, which means `O(1)`.
- The output string does not use extra space that grows with input `s`, as it's just a representation of `cnt`.

This leads to a space complexity of `O(1)` since both the `Counter` and the `cnt` array occupy constant space.

Thus, the given code has a time complexity of `O(n)` and a space complexity of `O(1)`.