1805. Number of Different Integers in a String

Easy | Hash Table | String

Problem Description

This problem involves a string `word` that is composed of both digits and lowercase English letters. The goal is to transform this string in a way that all non-digit characters are replaced by spaces, which effectively segments any clusters of digits into separate numbers. After this replacement, you'll have a collection of separate numbers that might include leading zeroes.

The task is now to count how many *different* integers you get from these numbers. The important point to note here is that integers with leading zeros should be considered the same as those without leading zeros. For instance, "001" and "1" should be counted as the same integer. The output should be the total number of unique integers you can find after performing the operations specified.

Intuition

When attacking this problem, we must focus on two main tasks:

1. Segregating the numbers from the mix of digits and letters.
2. Ensuring that any leading zeros are ignored to avoid counting the same number multiple times.

The intuitive approach here is to iterate through the string, identifying digits and collecting consecutive digits into whole numbers. Whenever we encounter a non-digit character, we treat that as a delimiter indicating the end of a number.

While iterating over the string and finding these numbers, we also need to deal with leading zeros. The solution handles this efficiently by first going through any '0's before finishing collecting the number and only then adding it to a set. A set is the ideal data structure here because it naturally avoids duplication. Whenever we encounter a sequence of digits, we slice that part out of the string and add it to our set, and the set ensures that it only contains unique numbers.

Solution Approach

The solution to the problem uses a simple yet effective algorithm that employs a `set` data structure and a single pass through the string to identify and store unique integers. Here's step-by-step implementation details:

1. **Initialize a Set**: A set named `s` is created to store integers uniquely encountered during the iteration. In Python, a set automatically handles duplicates, only storing unique values.
2. **Iterate Over the String**: The algorithm uses a `while` loop to iterate over the entire string, indexed by `i`. Using `while` is more flexible here than a `for` loop because we may need to increment `i` by more than one to skip over the digits or leading zeros.
3. **Find Integers**: Inside the loop, whenever a digit is encountered, the following steps are taken:
   - Skip over Leading Zeros: First, while the current character is '0', increment `i` to remove leading zeros from consideration.
   - Collect the Number: Then set a secondary index `j` to `i` and increment `j` until a non-digit character is found, which signifies the end of the current integer.
   - Add the Integer to the Set: With the start and end of an integer determined, a slice of `word` from `i` to `j` is taken (which is our integer without leading zeros), and added to the set `s`.
   - Update the Primary Index: Lastly, set `i` equal to `j`, as we've finished processing the current number and need to continue scanning the rest of the string.
4. **Increment the Index**: If the current character is not a digit, just increment `i` by one to move to the next character.
5. **Return Unique Count**: After the while loop is done, the algorithm returns the size of the set `s` (`len(s)`), which represents the count of unique integers found in the string.

By using a set and carefully iterating over the string, the solution efficiently filters out non-digit characters, handles leading zeros, and ensures that each integer is considered only once, despite how many times it appears in the string.

Example Walkthrough

Let's take a simple example string `word` equal to `"a10b00100c01"`. We aim to find out how many different integers are in the string after converting non-digit characters to spaces and considering integers with leading zeros as the same.

Here is how the solution approach applies to this example:

1. **Initialize a Set**: We create an empty set `s` to store unique integers.
2. **Iterate Over the String**: We start with `i = 0`. Since `word[0]` is 'a', a letter, we just increment `i` to 1.
3. **Find Integers**: When `i = 1`, we find '1', which is a digit. At this point:
   - Skip over Leading Zeros: There are no leading zeros, so we do nothing.
   - Collect the Number: We set `j` to 1 and increment `j` until we find a non-digit character. In this case, `j` becomes 3 when it reaches 'b'.
   - Add the Integer to the Set: We add `word[1:3]` (which is "10") to set `s`.
   - Update the Primary Index: We set `i` equal to `j` (i.e., `i = 3`).
4. **Increment the Index**: At `i = 3`, we have 'b', a non-digit. We increment `i` by one to 4.
5. **Find Integers**: Now, `i = 4`, and we find '0', a digit:
   - Skip over Leading Zeros: We increment `i` while the character is '0'. Now `i = 6`, at the next non-zero digit.
   - Collect the Number: We set `j` to 6 and increment `j` until we find a non-digit character. This happens immediately as 'c' at index 6 is a letter.
   - No number to add to the Set: Since `i` and `j` are the same, we don't have a number slice to add to the set `s`.
   - Update the Primary Index: `i` remains equal to `j` (i.e., `i = 6`).
6. **Increment the Index**: As `i = 6` and `word[i]` is 'c', we increment `i` by one to 7.
7. **Find Integers**: At `i = 7`, we find '0':
   - Skip over Leading Zeros: We increment `i` to skip over zeros and reach the non-zero digit '1' at `i = 9`.
   - Collect the Number: We set `j` to 9 and increment `j`. Since the end of the string is reached, `j` becomes the end of the string.
   - Add the Integer to the Set: We add `word[9:end]` (which is "1") to the set `s`.
   - Update the Primary Index: We set `i` to the end of the string because we've reached the end.
8. **Return Unique Count**: The iteration is finished. Our set `s` contains `{"10", "1"}`. The count of unique integers is the size of the set `s`, which is `2`.

The different integers we have found are `10` and `1`. Despite the multiple representations of the number one, such as "01", "001", "0001" etc., they are all counted as a single unique integer `1`.

Thus, the algorithm successfully arrives at the correct answer: there are 2 different integers in the string `"a10b00100c01"`.

Solution Implementation

Python

```python
class Solution:
    def numDifferentIntegers(self, word: str) -> int:
        # Initialize a set to store unique integers found in the string
        unique_integers = set()

        # Initialize index and get the length of the word
        index, length = 0, len(word)

        # Iterate over each character in the word
        while index < length:
            # Check if the current character is a digit
            if word[index].isdigit():
                # Skip leading zeros
                while index < length and word[index] == '0':
                    index += 1
                # Mark the start of the non-zero sequence
                start = index
                # Find the end of the continuous digit sequence
                while index < length and word[index].isdigit():
                    index += 1
                # Add the non-zero starting sequence of digits to the set
                unique_integers.add(word[start:index])
            # Move to the next character
            index += 1

        # Return the count of unique integers found
        return len(unique_integers)
```

Java

```java
class Solution {

    // Method to count the number of different integers in a string
    public int numDifferentIntegers(String word) {
        // Set to store unique integers represented as strings
        Set<String> uniqueNumbers = new HashSet<>();
        int length = word.length(); // Length of the input string

        // Loop through each character in the string
        for (int i = 0; i < length; ++i) {
            // Check if the current character is a digit
            if (Character.isDigit(word.charAt(i))) {
                // Skip leading zeros for the current number
                while (i < length && word.charAt(i) == '0') {
                    ++i;
                }

                // Starting index of the non-zero digit
                int startIndex = i;
                // Find the end index of the contiguous digits
                while (startIndex < length && Character.isDigit(word.charAt(startIndex))) {
                    ++startIndex;
                }

                // Add the integer (without leading zeros) to the set
                uniqueNumbers.add(word.substring(i, startIndex));

                // Move to the position after the current number
                i = startIndex;
            }
        }

        // Return the size of the set, which is the number of unique integers
        return uniqueNumbers.size();
    }
}
```

C++

```cpp
#include <string>
#include <unordered_set>

class Solution {
public:
    // Function to count the number of distinct integers in a string
    int numDifferentIntegers(std::string word) {
        std::unordered_set<std::string> distinctNumbers; // Set to store unique numbers as strings
        int length = word.size(); // Size of the input string

        for (int i = 0; i < length; ++i) {
            // If the current character is a digit, start processing the number
            if (isdigit(word[i])) {
                // Skip leading zeros
                while (i < length && word[i] == '0') ++i;

                // Find the end of the current number
                int start = i;
                while (start < length && isdigit(word[start])) ++start;

                // Insert the number as a substring into the set, ensuring uniqueness
                distinctNumbers.insert(word.substr(i, start - i));

                // Move to the end of the current number to continue the outer loop
                i = start;
            }
        }

        // Return the count of unique integers found in the string
        return distinctNumbers.size();
    }
};
```

TypeScript

```typescript
function numDifferentIntegers(word: string): number {
    // Use a regular expression to replace all non-digit characters with a single space
    const wordWithSpaces = word.replace(/\D+/g, ' ');

    // Trim leading and trailing spaces and split the string into an array on spaces
    const rawNumbers = wordWithSpaces.trim().split(' ');

    // Filter out any empty strings that may result from multiple adjacent non-digit characters
    const filteredNumbers = rawNumbers.filter(value => value !== '');

    // Remove leading zeros from each number string
    const cleanedNumbers = filteredNumbers.map(value => value.replace(/^0+/g, ''));

    // Create a Set to hold unique numbers and determine its size
    const uniqueNumbers = new Set(cleanedNumbers);

    return uniqueNumbers.size;
}


class Solution:
    def numDifferentIntegers(self, word: str) -> int:
        # Initialize a set to store unique integers found in the string
        unique_integers = set()

        # Initialize index and get the length of the word
        index, length = 0, len(word)

        # Iterate over each character in the word
        while index < length:
            # Check if the current character is a digit
            if word[index].isdigit():
                # Skip leading zeros
                while index < length and word[index] == '0':
                    index += 1
                # Mark the start of the non-zero sequence
                start = index
                # Find the end of the continuous digit sequence
                while index < length and word[index].isdigit():
                    index += 1
                # Add the non-zero starting sequence of digits to the set
                unique_integers.add(word[start:index])
            # Move to the next character
            index += 1

        # Return the count of unique integers found
        return len(unique_integers)
```

Time and Space Complexity

The given Python code defines a method to count the number of distinct integers in a string. Let's analyze both time and space complexity:

Time Complexity

The time complexity of the code can be broken down as follows:

1. Loop through each character in the string: $O(n)$, where n is the length of the string `word`.
2. Nested loop to skip leading zeros: In the worst case, this occurs once for each digit, contributing $O(n)$.
3. Nested loop to identify the digits of the integer: This also occurs once for each digit, contributing $O(n)$.

Although there are nested loops, each character is processed only once. Thus, the overall time complexity is $O(n)$.

Space Complexity

The space complexity is determined by the extra space used:

1. A set `s` is used to store distinct integers. In the worst case, each character could be a different non-zero digit, resulting in a set size of $O(n)$.
2. Substrings representing integers are added to the set, but the total length of all unique integer substrings cannot exceed n.

Hence, the space complexity is $O(n)$.

Overall, both the time complexity and the space complexity of the method are $O(n)$, where n is the length of the input string `word`.