2042. Check if Numbers Are Ascending in a Sentence

Easy String

Leetcode Link

Problem Description

The problem asks to determine if all numbers in a given sentence are strictly increasing. A sentence is defined as a list of tokens, which are either a positive number or a word, separated by single spaces. There are no leading zeros in numbers and no leading or trailing spaces in the sentence. The numbers must increase from left to right, meaning that each number should be smaller than the number that follows it. The task is to return `true` if the condition of strictly increasing numbers is met, otherwise return `false`.

Intuition

The intuition behind solving this problem is to iterate over the tokens in the sentence and keep track of the last number encountered. We split the sentence into tokens based on spaces, and then scan each token. If a token is a number (we can check this by looking at the first character of the token), we then compare it with the previous number we have seen (if any). To verify that the numbers are strictly increasing, we need to ensure that the current number is greater than the last number seen. If at any point we encounter a number that is not greater than the previous one, we return `false` as the sequence is no longer strictly increasing. If we successfully go through all the numbers without violating the condition, we return `true`, which means all numbers are strictly increasing.

Solution Approach

The implementation of the solution uses a simple iteration approach. The only data structure used here is an integer variable `pre` to store the previously seen number, which initially is set to `0`, being smaller than any positive number as defined in the problem. We employ Python's built-in `str.split()` method to break the sentence `s` into tokens based on space as a delimiter.

The solution involves the following steps:

1. We start by splitting the sentence `s` into tokens by using the `split` method. Each token is either a word or a number.
2. We iterate over each token `t` in the sentence:
   - We check if the token is a number by looking at the first character of the token with `t[0].isdigit()`.
   - If it is a number, we convert the token to an integer `cur` with `int(t)`.
   - We then check if `cur` is less than or equal to the previous number `pre`. If so, we immediately return `false` since the numbers must be strictly increasing.
   - If the condition is not met, we update `pre` to be the current number `cur`.
3. If we finish iterating through all the tokens without returning `false`, we return `true` as no condition was violated and therefore the numbers in the sentence are strictly increasing.

Here is the essence of the code, showing the straightforward loop and checks:

```
pre = 0
for t in s.split():
    if t[0].isdigit():
        cur = int(t)
        if cur <= pre:
            return False
        pre = cur
return True
```

The code is efficient, running in `O(n)` time complexity – where `n` is the number of characters in the string – because it checks each token exactly once. Additionally, the space complexity is also `O(n)` due to the space taken by the split tokens. By using tuple unpacking with the walrus operator (`:=`), the code is concise and avoids a nested `if` statement.

Example Walkthrough

Let's illustrate the solution approach using a simple example:

Given the sentence `"cat 1 dog 2 fish 3 cow 5 lion 8"`

1. We start by splitting the sentence into tokens: `["cat", "1", "dog", "2", "fish", "3", "cow", "5", "lion", "8"]`.
2. Now, we initialize `pre = 0`, and iterate over the tokens:
   - The first token is "cat", which starts with a letter, so we ignore it.
   - The second token is "1", which starts with a digit. We convert "1" to integer `1` and compare it with `pre` (0). Since `1 > 0`, the condition is satisfied. We set `pre = 1`.
   - The next token "dog" is ignored since it doesn't start with a digit.
   - The following token is "2", which is again a number. We convert "2" to integer `2` and compare it with `pre` (1). Since `2 > 1`, the condition is satisfied. We set `pre = 2`.
   - This process continues for each number token.
3. Each number is greater than the last, so the condition of strictly increasing numbers is satisfied throughout.
4. After checking all the tokens, since we've found no violation of the condition, the function will return `True`.

In this example, the sentence follows the strictly increasing numerical order despite being interspersed with words. Thus, the output is `True`.

Python Solution

```python
class Solution:
    def areNumbersAscending(self, s: str) -> bool:
        # Initialize the previous number to a value that won't
        # be greater than any number encountered in the string.
        previous_number = -1

        # Split the string into individual tokens and iterate through them.
        for token in s.split():
            # Check if the first character of the token is a digit.
            if token[0].isdigit():
                # Convert the token to an integer.
                current_number = int(token)

                # If the current number is less than or equal to the previous number,
                # the numbers are not strictly ascending.
                if current_number <= previous_number:
                    return False

                # Update the previous number to the current one
                # for the next iteration's comparison.
                previous_number = current_number

        # If we've gone through all numbers without returning False,
        # the numbers are strictly ascending.
        return True
```

Java Solution

```java
class Solution {

    /**
     * Checks if the numbers in the string are in ascending order.
     *
     * @param str The string containing words and numbers.
     * @return {@code true} if the numbers appear in ascending order, {@code false} otherwise.
     */
    public boolean areNumbersAscending(String str) {
        // Previous number to compare with, initialized to the smallest possible value (0 is a valid number, so -1 is used)
        int previousNum = -1;

        // Split the input string on spaces
        String[] tokens = str.split(" ");

        // Iterate through each split token
        for (String token : tokens) {
            // Check if the first character of the token is a digit
            if (Character.isDigit(token.charAt(0))) {
                // Parse the integer value from the token
                int currentNum = Integer.parseInt(token);
                // Compare the current number with the previous one
                if (previousNum >= currentNum) {
                    // If the current number is not greater than the previous number, return false
                    return false;
                }
                // Update previous number to the current number
                previousNum = currentNum;
            }
        }

        // If all numbers were in ascending order, return true
        return true;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // This function checks if numbers in the sentence are strictly ascending.
    bool areNumbersAscending(std::string s) {
        int previousNumber = -1; // Initialize with -1 to handle the case when the first token is a number.
        std::istringstream stream(s); // Use istringstream to read the string token by token.
        std::string token;

        // Loop through each token in the input string.
        while (stream >> token) {
            // Check if the first character of the token is a digit, indicating it's a number.
            if (isdigit(token[0])) {
                int currentNumber = std::stoi(token); // Convert token to integer.

                // If current number is not greater than the previous number, return false.
                if (previousNumber >= currentNumber) {
                    return false;
                }

                // Update the previous number to the current number for next comparison.
                previousNumber = currentNumber;
            }
        }

        // If all numbers are in strictly ascending order, return true.
        return true;
    }
};
```

Typescript Solution

```typescript
function areNumbersAscending(s: string): boolean {
    // Initialize previousNumber with a value that is less than any other number.
    let previousNumber = -1;

    // Split the input string by spaces to check each word.
    for (const word of s.split(' ')) {
        // If the first character of the word is a digit,
        // the word might be a number or start with a number.
        if (!isNaN(word[0] as any)) {
            // Try to convert the word to a number.
            const currentNumber = Number(word);

            // If the conversion is successful and the currentNumber is not greater than previousNumber,
            // the sequence is not strictly ascending, so return false.
            if (currentNumber <= previousNumber) {
                return false;
            }

            // Update previousNumber to the currentNumber for the next iteration's comparison.
            previousNumber = currentNumber;
        }
    }

    // If the loop completes without returning false, the condition is met for an ascending sequence of numbers.
    return true;
}

```

Time and Space Complexity

Time Complexity:

The time complexity of the code is `O(n)`, where `n` is the length of the string `s`. This is because the `split` method runs in `O(n)`, splitting the string into words based on spaces, and the `for` loop iterates over each word once. The loop itself contains a constant-time check (`if t[0].isdigit()`) and a constant-time integer conversion (`int(t)`), so overall, each iteration of the loop adds a constant amount of work. Consequently, the total time taken is proportional to the number of words, which is at most proportional to the length of the input string `s`.

Space Complexity:

The space complexity of the code is `O(m)`, where `m` is the number of words in the string `s`. This is because the `split` method creates a list of all the words in the string, which requires space proportionate to the number of words. The `for` loop does not allocate any additional data structures that grow with the size of the input; it only uses a couple of variables to hold the current and previous numerical values. Note that in the worst case, every character could be a word (if they are all digits separated by spaces), making `m` roughly `n/2`, but this does affect the big-O notation and the space complexity remains linear with respect to the length of the input string.