1832. Check if the Sentence Is Pangram

Easy Hash Table String

Leetcode Link

Problem Description

A *pangram* is a special kind of sentence in which every letter of the English alphabet is used at least once. The task is to determine whether a given sentence meets the criteria to be a pangram. The input is a string named `sentence` which consists solely of lowercase English letters. The expected output is a boolean value: `true` if the given `sentence` is a pangram, and `false` otherwise.

Intuition

The intuition behind the solution is to leverage the fact that there are 26 letters in the English alphabet. To verify if a sentence is a pangram, we need to check that each of these letters appears at least once. Using a set is a smart approach because a set automatically filters out duplicate elements. When we convert the sentence into a set, any repeated letters are removed, leaving us with a set of unique characters.

By checking the number of unique characters (the length of the set), we can ascertain whether all 26 letters of the alphabet are present. If the unique character count is precisely 26, it means every letter of the alphabet is represented in the sentence, and hence, the sentence is a pangram. The provided solution code succinctly does this check in one line by comparing the length of the set derived from `sentence` against the number 26. If they match, `true` is returned; otherwise `false`.

Solution Approach

The solution's implementation is straightforward and utilizes Python's built-in data structure—a set—and its characteristics to solve the problem efficiently. The key points of the approach are:

1. **Use of a Set:** A set is chosen because it automatically handles the removal of duplicate characters. When we cast the `sentence` string to a set, all duplicate letters are removed, leaving us with only unique characters. This is done with `set(sentence)`.

2. **Count Unique Characters:** After creating a set of unique characters, we simply count how many unique characters are contained in it. This is achieved by calling `len(set(sentence))`. The `len` function returns the count of how many elements (in this case, unique letters) are in the set.

3. **Comparison with Alphabet Length:** The final step is to compare this count of unique characters to the total number of letters in the English alphabet, which is 26. If the sentence contains every letter of the alphabet at least once, then `len(set(sentence))` should be equal to 26.

4. **Return the Result:** The comparison `len(set(sentence)) == 26` will yield a boolean value (`True` or `False`). If the count is 26, it will return `True`, indicating that the sentence is a pangram. Otherwise, if any letter is missing, and the count is less than 26, the result will be `False`, indicating that the sentence is not a pangram.

5. **One-Line Solution:** The implementation combines these steps into a single line. By returning the result of the comparison directly, the solution becomes both concise and efficient.

Thus, the solution effectively leverages the properties of sets to eliminate redundant operations or the need for explicit loops, resulting in a lean and performant algorithm for checking whether a given sentence is a pangram.

Example Walkthrough

Let's take a sentence `sentence = "the quick brown fox jumps over a lazy dog"` and walk through the process to determine if it's a pangram:

1. **Use of a Set:** First, convert the sentence into a set of unique characters: `unique_chars = set(sentence)`. The set now contains each character from the sentence without any duplicates.

2. **Count Unique Characters:** Count how many unique characters are in the set: `unique_count = len(unique_chars)`. Assuming the sentence is indeed a pangram, the count should include 26 letters plus any additional unique characters like spaces or punctuation marks.

3. **Comparison with Alphabet Length:** Now compare this count of unique characters (excluding spaces and punctuation) with the English alphabet's 26 letters. Since we are only interested in the letters, if we had other characters, we would ignore them. But since the sentence is constructed with only letters and spaces, and we can ignore spaces in our count, a direct comparison can be made: `is_pangram = (unique_count == 26)`.

4. **Return the Result:** Finally, the result of the comparison will be a boolean value. If `unique_count` equals 26, `is_pangram` will be `True`, indicating that the sentence contains every letter of the English alphabet at least once. Otherwise, it will be `False`.

Following the given sentence, when we apply the solution, we find that after removing duplicates and excluding spaces, the set contains exactly 26 letters. So `len(set(sentence))` equals 26, and the sentence is confirmed to be a pangram, hence `True` is returned.

Python Solution

```python
class Solution:
    def checkIfPangram(self, sentence: str) -> bool:
        # A pangram is a sentence containing every letter of the alphabet at least once.
        # To check if a sentence is a pangram, we convert the sentence into a set of characters.
        # A set in Python is a collection data type that is unordered, mutable, and does not allow duplicates.
        # Therefore, converting the sentence into a set removes any duplicate characters.

        unique_characters = set(sentence)

        # We then check if the number of unique characters is 26, which is the number of letters in the English alphabet.
        # If there are 26 unique characters, then the sentence must be a pangram and we return True.
        # Otherwise, we return False as the sentence is missing one or more letters.

        is_pangram = len(unique_characters) == 26

        return is_pangram

```

Java Solution

```java
class Solution {
    // Checks if the input string is a pangram (contains all letters of the alphabet at least once)
    public boolean checkIfPangram(String sentence) {
        int alphabetMask = 0; // Used to track the presence of each letter

        // Iterate over each character in the sentence
        for (int i = 0; i < sentence.length(); i++) {
            // Update the alphabetMask by setting the bit corresponding to the current character
            // The bit position is determined by subtracting the ASCII value of 'a' from the character
            alphabetMask |= 1 << (sentence.charAt(i) - 'a');
        }

        // The sentence is a pangram if alphabetMask has 26 bits set (one for each letter of the alphabet)
        // (1 << 26) - 1 creates a mask with 26 bits set to 1
        return alphabetMask == (1 << 26) - 1;
    }
}

```

C++ Solution

```cpp
#include <string>

class Solution {
public:
    bool checkIfPangram(const std::string& sentence) {
        // Initialize a variable to keep track of the presence of each letter
        // by using a bit mask. Each bit from 0 to 25 will correspond to
        // the letters 'a' through 'z'.
        int letterMask = 0;

        // Iterate through each character in the sentence
        for (const char& currentChar : sentence) {
            // Calculate the bit number that represents the current character
            // by subtracting 'a' from it, and then set that bit in letterMask
            // using bitwise OR and bit shift left operations.
            letterMask |= 1 << (currentChar - 'a');
        }

        // Check if letterMask has all 26 bits set, which means all letters
        // from 'a' to 'z' are present. A mask with all 26 bits set represents
        // the number (2^26) - 1. If letterMask is equal to this number, then
        // the sentence is a pangram.
        return letterMask == (1 << 26) - 1;
    }
};

// Usage example:
// Solution sol;
// bool isPangram = sol.checkIfPangram("the quick brown fox jumps over a lazy dog");

```

Typescript Solution

```typescript
function checkIfPangram(sentence: string): boolean {
    // Initialize a variable to track the letters found in the sentence
    // Each bit of 'mark' represents one letter of the alphabet
    let mark = 0;

    // Iterate over each character in the sentence
    for (const char of sentence) {
        // Calculate the alphabet index relative to 'a'
        // 'a' will be 0, 'b' will be 1 and so on
        const alphabetIndex = char.charCodeAt(0) - 'a'.charCodeAt(0);

        // Set the bit at the alphabet index to 1
        // This uses a bitwise OR assignment to mark the presence of the character
        mark |= 1 << alphabetIndex;
    }

    // Check if all 26 bits are set to 1
    // (1 << 26) creates a number with 26 0s followed by a 1
    // Subtracting 1 from that number flips all 26 0s to 1s
    // Comparing 'mark' to this value verifies if all letters of the alphabet are present
    return mark === (1 << 26) - 1;
}

```

Time and Space Complexity

The time complexity of the provided code is `O(N)`, where `N` is the length of the input string `sentence`. This is because constructing a set from the string requires iterating over each character in the string once to produce a set of unique characters.

The space complexity of the code is also `O(N)`, assuming that in the worst case, there are no duplicate characters in the string. The size of the set data structure would grow linearly with the size of the input string up to 26, as there are a maximum of 26 different lower-case English letters. The space complexity becomes constant `O(1)` when considering that we have a fixed upper limit on the size of the set (26 characters), if we evaluate the space complexity as a function of the character set size rather than the string length.