3019. Number of Changing Keys

Easy String

Problem Description

In this problem, we're given a 0-indexed string `s` that represents the sequence of keys typed by a user. The string could consist of lowercase and uppercase letters from 'a' to 'z'. Our goal is to determine the number of key changes that the user has made. A key change is defined as typing a character that is different from the immediately preceding character, regardless of whether the characters are in different cases or the same (for example, typing 'a' after 'A' is not considered a change). Essentially, we need to count the instances where one character followed by another is different when both characters are converted to lowercase.

Modifiers like 'shift' or 'caps lock' which are used to change the case of letters are not considered key changes in themselves. This means that just switching from uppercase to lowercase or vice versa does not count as a change. The user has to move to a different letter altogether for it to be considered a change of keys.

Intuition

To arrive at the solution for this problem, we need an efficient way to compare consecutive characters in the string and count only the instances where a letter is followed by a different letter. Here's the intuition behind our approach:

1. We iterate over the string `s` to examine each character in sequence.
2. We compare each character with the next one, ignoring case (we use the lowercase of both characters for comparison).
3. If the characters differ, it means the user had to change the key, so we increment a counter.
4. We continue this process for the entire string, and the sum of increments in the counter gives us the total number of key changes.

Solution Approach

Following the intuition from before, the implementation involves traversing the string and performing a simple comparison for each pair of consecutive characters. Here's how it breaks down:

Data Structures and Patterns

We use the `pairwise` function, which is typically used to generate a new iterable by returning pairs of elements from the input iterable. In Python, it would provide us with an iterator over pairs of consecutive items. Here, `pairwise(s)` will return (s[0], s[1]), (s[1], s[2]), ... (s[n-2], s[n-1]) where `n` is the length of the string.

Algorithms

The solution makes use of the following simple algorithm:

1. **Convert Characters to Lowercase and Compare**: The `pairwise` function gives us each pair of consecutive characters `a` and `b` as tuples. For each tuple, we convert `a` and `b` to lowercase and then check if they are not equal to each other using `a.lower() != b.lower()`.
2. **Count Key Changes**: If the lowercase versions of `a` and `b` are not equal, it indicates a key change. Using a generator expression inside `sum`, we iterate over all pairs, convert to lowercase, compare, and count these instances directly.
3. **Summation**: `sum` is used to add up all the `True` values from the generator expression, where each `True` represents a different comparison and hence a key change.

The algorithm completes in a single pass over the string, which gives us a time complexity of O(n), where n is the length of the string.

```
def countKeyChanges(self, s: str) -> int:
    return sum(a.lower() != b.lower() for a, b in pairwise(s))
```

This single line of Python code is all that's needed to implement the complete algorithm, leveraging the power of Python's built-in functions and concise syntax.

Efficiency

Since we're performing one iteration of the string, and each operation within that iteration is O(1), the overall time complexity is O(n), making this approach efficient and scalable for large strings. There are no complex data structures involved, and we only require additional space for the lowercase character comparisons, which is constant space O(1), thus making the space complexity of this algorithm also very efficient.

Example Walkthrough

Let's consider a small example to illustrate the solution approach, using the string `s = "AabB"`. We want to find out the number of key changes the user has made, recalling that a 'key change' is defined as typing a distinct character than the previous one, case-insensitively.

Following the provided solution approach:

1. We will use the `pairwise` function to create pairs of consecutive characters from our string `s`, resulting in the pairs: ('A', 'a'), ('a', 'b'), and ('b', 'B'). Here, `pairwise(s)` basically generates an iterator that would give us these pairs when looped over.
2. For each pair, we convert each element to its lowercase version and compare them:
   - Compare `A` with `a`: Since `A.lower()` == `a.lower()`, there's no key change.
   - Compare `a` with `b`: Since `a.lower()` != `b.lower()`, we have a key change.
   - Compare `b` with `B`: Since `b.lower()` == `B.lower()`, there's no key change.
3. We use a generator expression within `sum` to count the key changes. In our case, the pairs would give us `[False, True, False]` when comparing their lowercased versions. We are only interested in the instances where the comparison yields `True`.
4. Finally, we sum up the `True` values using `sum()`, which represents the total key changes. For our example, we get `sum([False, True, False])`, which simplistically is just 0 + 1 + 0, equaling 1.

Therefore, using the given method, the user has made 1 key change while typing the string "AabB".

The corresponding Python code using the provided algorithm would process this example as follows:

```
from itertools import pairwise

def countKeyChanges(s: str) -> int:
    return sum(a.lower() != b.lower() for a, b in pairwise(s))

# Example string
s = "AabB"
# Expected output: 1
print(countKeyChanges(s))
```

When we execute this code with our example string `s = "AabB"`, the output will be `1`, which matches our manual calculation.

Solution Implementation

Python

```
from itertools import pairwise  # pairwise is from itertools in Python 3.10 and later. For earlier versions, you may need to def

class Solution:
    def countKeyChanges(self, keyboard_sequence: str) -> int:
        """
        Count the number of key changes in a given keyboard sequence.

        :param keyboard_sequence: A string representing the sequence of key presses.
        :return: An integer representing the number of key changes.
        """

        # Initialize a counter for key changes
        key_changes = 0

        # Use the pairwise function to create pairs of consecutive elements
        for current_key, next_key in pairwise(keyboard_sequence):
            # Check if the lowercase versions of the consecutive keys are different
            if current_key.lower() != next_key.lower():
                # If so, increment the key changes counter
                key_changes += 1

        # Return the total number of key changes
        return key_changes

# Example usage:
# solution = Solution()
# result = solution.countKeyChanges("aAbBBbC")
# print(result)  # Output would be the number of key changes.


Key points in this rework:
- The method name `countKeyChanges` remains unchanged as per the instructions.
- The `s` parameter is renamed to `keyboard_sequence` to provide more clarity on what it represents.
- A comment section has been added before the method to explain the functionality, inputs, and output.
- Added inline comments to explain each step of the code for better understanding.

Remember that the `pairwise` function was added in Python 3.10. If you're using a version of Python prior to 3.10, you would have

```python
def pairwise(iterable):
    "s -> (s0,s1), (s1,s2), (s2, s3), ..."
    a, b = itertools.tee(iterable)
    next(b, None)
    return zip(a, b)
```
```

Java

```
class Solution {
    // Method to count the number of key changes in the input string
    public int countKeyChanges(String s) {
        // Initialize the count of key changes to zero
        int keyChangeCount = 0;

        // Iterate over the characters in the string starting from the second character
        for (int i = 1; i < s.length(); ++i) {
            // Compare current and previous characters in a case-insensitive manner
            if (Character.toLowerCase(s.charAt(i)) != Character.toLowerCase(s.charAt(i - 1))) {
                // Increment the number of key changes when the current and previous characters differ
                keyChangeCount++;
            }
        }

        // Return the total count of key changes
        return keyChangeCount;
    }
}
```

C++

```
#include <algorithm> // Include algorithm header for transform
#include <string>    // Include string header for string class

class Solution {
public:
    // This function counts the number of times the key changes (case-insensitive).
    int countKeyChanges(std::string s) {
        // Convert the entire string to lower case to make comparison case-insensitive
        std::transform(s.begin(), s.end(), s.begin(),
                       [](unsigned char c) { return std::tolower(c); });

        // Initialize a counter for the number of key changes
        int count = 0;

        // Iterate through the string starting from the second character
        for (size_t i = 1; i < s.size(); ++i) {
            // If the current character is different from the previous one,
            // increment the count of key changes
            count += (s[i] != s[i - 1]);
        }

        // Return the total count of key changes
        return count;
    }
};
```

TypeScript

```
function countKeyChanges(inputString: string): number {
    // Convert the input string to lowercase for a case-insensitive comparison
    inputString = inputString.toLowerCase();

    // Initialize a counter for the number of key changes
    let changeCount = 0;

    // Iterate over the characters of the string, starting from the second character
    for (let index = 1; index < inputString.length; ++index) {
        // Compare the current character with the previous one
        if (inputString[index] !== inputString[index - 1]) {
            // If they differ, increment the change count
            ++changeCount;
        }
    }

    // Return the total number of key changes in the string
    return changeCount;
}


from itertools import pairwise  # pairwise is from itertools in Python 3.10 and later. For earlier versions, you may need to define i

class Solution:
    def countKeyChanges(self, keyboard_sequence: str) -> int:
        """
        Count the number of key changes in a given keyboard sequence.

        :param keyboard_sequence: A string representing the sequence of key presses.
        :return: An integer representing the number of key changes.
        """

        # Initialize a counter for key changes
        key_changes = 0

        # Use the pairwise function to create pairs of consecutive elements
        for current_key, next_key in pairwise(keyboard_sequence):
            # Check if the lowercase versions of the consecutive keys are different
            if current_key.lower() != next_key.lower():
                # If so, increment the key changes counter
                key_changes += 1

        # Return the total number of key changes
        return key_changes

# Example usage:
# solution = Solution()
# result = solution.countKeyChanges("aAbBBbC")
# print(result)  # Output would be the number of key changes.


Key points in this rework:
- The method name `countKeyChanges` remains unchanged as per the instructions.
- The `s` parameter is renamed to `keyboard_sequence` to provide more clarity on what it represents.
- A comment section has been added before the method to explain the functionality, inputs, and output.
- Added inline comments to explain each step of the code for better understanding.

Remember that the `pairwise` function was added in Python 3.10. If you're using a version of Python prior to 3.10, you would have to

```python
def pairwise(iterable):
    "s -> (s0,s1), (s1,s2), (s2, s3), ..."
    a, b = itertools.tee(iterable)
    next(b, None)
    return zip(a, b)
```
```

Time and Space Complexity

The time complexity of the provided code is `O(n)`, where `n` is the length of the string `s`. This is because the `pairwise` function generates pairs of consecutive characters from the string and the `sum` function goes through each pair only once in a single pass. Hence, the number of operations is proportional to the number of characters in the string.

The space complexity of the code is `O(1)`. This is due to the fact that the `pairwise` function does not create a separate list of pairs but instead generates them on-the-fly, and the summation computation does not require additional space that grows with the input size. Only a constant amount of extra memory is used for the iterator and the count variable within the sum, regardless of the size of the string `s`.