2138. Divide a String Into Groups of Size k

Easy String Simulation

Leetcode Link

Problem Description

This problem is about partitioning a string `s` into several groups of characters. Each group must contain exactly `k` characters. The partitioning follows a set procedure:

- We start from the beginning of the string and create groups by sequentially taking `k` characters for each until we reach the end of the string.
- If the remaining characters at the end of the string are fewer than `k`, we fill the last group with a fill character `fill` to make sure it also has exactly `k` characters.

The constraint is that after removing any `fill` characters added to the last group and then concatenating all groups, we should get the original string `s`.

The goal is to return an array of strings, where each element is one of the groups created by this procedure.

Intuition

The solution relies on a straightforward approach:

- Iteratively process the string in increments of `k` characters. For every `k` characters, we take a substring from the original string `s` and add it to the result list.
- We use a Python list comprehension to perform this operation concisely. The list comprehension iterates over the range of indices starting from 0 up to the length of the string `s`, skipping `k` indices at a time.
- For each iteration, we take a slice of the string `s` from the current index `i` up to `i + k`. If `i + k` exceeds the length of the string `s`, Python's slice operation will just return the substring up to the end of the string, which would be less than `k` characters.
- To ensure that each substring in the result list has exactly `k` characters, we use the `.ljust()` string method. This method pads the string on the right with the `fill` character until the string reaches a specified length (`k` in our case). If the substring already has `k` characters, `.ljust()` leaves it unchanged.
- This solution is efficient and only goes through the string once. It also takes care of the edge case where the last group is shorter than `k` characters without needing additional conditional checks.

By leveraging Python's string slicing and the `.ljust()` method, we achieve an elegant and efficient solution that directly constructs the desired array of group strings.

Solution Approach

In the given problem, we have a simple yet efficient solution that efficiently partitions the string into the required groups. The solution uses Python list comprehension, string slicing, and the `str.ljust` method. Let's walk through the key parts of the implementation:

- **List Comprehension**: This is a concise way to create lists in Python. It allows for the construction of a list by iterating over some sequence and applying an expression to each item in the sequence. In our case, we generate a list that contains each group of the partitioned string `s`.
- **String Slicing**: A vital feature of Python that allows us to extract a specific portion of a string using the syntax `string[start:end]`, where `start` is the index of the first character inclusive and `end` is the index of the last character exclusive. If `end` extends beyond the length of the string, Python simply returns the substring up to the end without raising an error.
- **`str.ljust(k, fill)` Method**: This string method left-justifies the string, using `fill` (a single character) as the fill character, until the whole string is of length `k`. If the string is already `k` characters or longer, no filling occurs. This method is perfect for ensuring that each group is exactly `k` characters long, even the last group if it's originally shorter than `k` characters.

The implementation is straightforward:

1. We iterate over the indices of the string `s` with the `range()` function, starting from `0` and ending at `len(s)`, taking steps of `k` at a time. This ensures that each iteration corresponds to a group of `k` characters.
2. For each index `i`, we slice the string `s` from `i` to `i+k`. This gives us a substring of `s` which is at most `k` characters long.
3. We apply the `ljust()` method to each substring with the specified `fill` character to ensure it is `k` characters long.

After iterating over all possible starting indices that are `k` characters apart, we collect all the modified substrings into a list. This list represents the groups after the partitioning process and is the final output of the function.

The code for creating each group looks like this:

```
[s[i : i + k].ljust(k, fill) for i in range(0, len(s), k)]
```

As a result, we get a list of strings, each representing a group of `s` that is exactly `k` character long, where the last group is filled with `fill` character if needed.

This approach is both simple to understand and effective at solving the problem, utilizing core Python string manipulation features to achieve the desired result with minimal code.

Example Walkthrough

Let's assume our input string `s` is "abcdefghi" and we are to partition this string into groups of `k = 3` characters. We also have a fill character `fill = "x"` to use if necessary.

According to the problem description, we would start at the beginning of the string and take groups of `k` characters in sequence. Our groups would be "abc," "def," and "ghi." Since "ghi" has exactly 3 characters, there is no need to use the `fill` character in this example.

We can illustrate the solution approach with the following steps:

1. Start at the beginning of the string, with `i = 0`.
2. Take the substring from `i` to `i + k`, which translates to `s[0:3]`. This gives us "abc".
3. Move to the next `k` characters, increment `i` by `k` to `i = 3`, and take the substring `s[3:6]`, which gives us "def".
4. Continue the process by setting `i` to 6 and taking the substring `s[6:9]`, yielding "ghi".

We now have the strings "abc," "def," and "ghi". All of these are of length `k`, so the `ljust` method is not necessary in this case.

However, if we had a different string, let's say `s` equals "abcdefg", with the same value for `k`, we'd have an additional step for the last group:

1. Following the previous steps, we would get "abc" and "def".
2. The next `i` is 6. The substring `s[6:9]` would result in "g". Since this group is less than `k` characters, we need to pad it with the `fill` character "x" to get "gxx".

Thus, the use of `s[i : i + k].ljust(k, fill)` in our list comprehension is to handle cases like this, ensuring the last group always has `k` characters. If the last group is originally shorter than `k` characters, it gets padded with `fill` until it's `k` characters long.

So, the partitioned groups for "abcdefg" will be:

- "abc"
- "def"
- "gxx"

The list comprehension puts this all together elegantly, completing the task in a single line of code.

Python Solution

```python
# Solution class containing the method to divide and fill a string.
class Solution:
    def divideString(self, s: str, k: int, fill: str) -> list[str]:
        # This method takes in a string `s`, a chunk size `k`, and a fill character `fill`.
        # It divides the string `s` into chunks of size `k` and if the last chunk
        # is smaller than `k`, it fills it with the `fill` character until it is of size `k`.
        # The method returns a list of these chunks.

        # Initialize an empty list to store the chunks of the string.
        result = []

        # Loop through the string in steps of `k` to get each chunk.
        for i in range(0, len(s), k):
            # Extract the chunk from the string starting at index `i` up to `i + k`.
            chunk = s[i : i + k]

            # If the length of the chunk is less than `k`, fill it with the fill character.
            # The `ljust` method is used to fill the chunk with the given character until it reaches the desired length.
            filled_chunk = chunk.ljust(k, fill)

            # Append the filled chunk to the `result` list.
            result.append(filled_chunk)

        # Return the list of chunks.
        return result
```

Java Solution

```java
class Solution {
    public String[] divideString(String input, int partitionSize, char fillCharacter) {
        // Determine the length of the input string.
        int inputLength = input.length();

        // Calculate the required number of partitions.
        int totalPartitions = (inputLength + partitionSize - 1) / partitionSize;

        // Initialize the answer array with the calculated size.
        String[] partitions = new String[totalPartitions];

        // If the input string is not a multiple of partition size, append fill characters to make it so.
        if (inputLength % partitionSize != 0) {
            input += String.valueOf(fillCharacter).repeat(partitionSize - inputLength % partitionSize);
        }

        // Loop through each partition, filling the partitions array with substrings of the correct size.
        for (int i = 0; i < partitions.length; ++i) {
            // Calculate the start and end indices for the substring.
            int start = i * partitionSize;
            int end = (i + 1) * partitionSize;

            // Extract the substring for the current partition and assign it to the partitions array.
            partitions[i] = input.substring(start, end);
        }

        // Return the final array of partitions.
        return partitions;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function to divide a string into substrings of equal length k,
    // If the length of string is not a multiple of k, fill the last substring with the 'fill' char.
    vector<string> divideString(string s, int k, char fill) {
        int stringLength = s.size();
        // If string length is not a multiple of k, append 'fill' characters
        if (stringLength % k) {
            for (int i = 0; i < k - stringLength % k; ++i) {
                s.push_back(fill);
            }
        }
        vector<string> substrings;
        // Divide the string into substrings of length k
        for (int i = 0; i < s.size() / k; ++i) {
            substrings.push_back(s.substr(i * k, k));
        }
        return substrings; // Return the vector of substrings
    }
};
```

Typescript Solution

```typescript
// Declare an array to store the resulting substrings
const substrings: string[] = [];

// Function to divide a string into substrings of equal length `k`
// If the length of string `s` is not a multiple of `k`, fill the last substring with the `fill` character
function divideString(s: string, k: number, fill: string): string[] {
    let stringLength: number = s.length;

    // If string length is not a multiple of `k`, append `fill` characters
    while (stringLength % k !== 0) {
        s += fill;
        stringLength++;
    }

    // Divide the string into substrings of length `k`
    for (let i = 0; i < s.length; i += k) {
        substrings.push(s.substring(i, i + k));
    }

    // Return the array of substrings
    return substrings;
}
```

Time and Space Complexity

Time Complexity:

The time complexity of the provided code is mainly determined by the list comprehension. It consists of a `for` loop that runs every `k` characters across the string `s` and an `ljust` operation within each iteration. Here, `n` is the length of the input string `s`.

- The `for` loop iterates `n/k` times because it jumps `k` characters in each iteration.
- The `ljust` operation potentially runs in `O(k)` time in the worst case, when the last segment of the string is less than `k` characters and needs to be filled with the fill character.

Therefore, we consider all iterations to get the time complexity: `n/k` iterations * `O(k)` per `ljust` operation = `O(n/k * k)` = `O(n)`.

So, the overall time complexity of the code is `O(n)`.

Space Complexity:

- The space complexity is determined by the space required to store the output list.
- Each element of the list is a string of `k` characters (or fewer if `fill` characters are added), so the space taken by each element is `O(k)`.
- Since there are `n/k` elements in the list, the total space for the list is `O(n/k * k)` = `O(n)`.

Additionally, the space taken by the input string `s` does not count towards the space complexity, as it is considered given and not part of the space used by the algorithm.

Thus, the overall space complexity of the code is `O(n)`, where `n` is the length of the input string `s`.