[Click on the logo to find **Problem Statement**]

Intuition

The provided code aims to group anagrams from a given list of strings (`strs`). The algorithm employs a dictionary (`anagram_groups`) to efficiently organize the strings based on their sorted representations, treating anagrams as having the same signature. The algorithm iterates through each word in the input list, sorts its characters to create a unique signature, and checks if this signature is already a key in the dictionary. If not, a new entry is added, and the original word is appended to the corresponding list. The result is a dictionary where keys represent anagram signatures, and values are lists of words that share the same signature. Finally, the algorithm returns a list of these grouped anagram lists. This approach ensures an efficient O(N * K * log(K)) time complexity, where N is the number of words and K is the maximum length of any word, making it practical for processing large sets of anagrams. The space complexity is O(N * K), as space is required to store the sorted words and their corresponding anagram groups in the dictionary.

Approach

1. **Initialization:**
    - The algorithm starts by initializing an empty dictionary named `anagram_groups` . This dictionary will be used to store anagram signatures as keys and lists of anagrams as values.
2. **Empty List Check:**
    - The algorithm checks if the input list of strings (`strs`) is empty. If it is, the function immediately returns the empty list, as there are no anagrams to group.
3. **Iterating Through Words:**
    - For each word in the input list `strs` , the algorithm proceeds to the next steps.
4. **Sorting Characters:**
    - The characters of each word are sorted to create a unique signature for anagrams. This is done using the `sorted` function and joining the characters back into a string.
5. **Dictionary Check and Update:**

- The algorithm checks if the sorted signature is already a key in the `anagram_groups` dictionary.
- If the signature is not present, a new entry is added to the dictionary with the sorted signature as the key and an empty list as the initial value.

6. **Appending to Anagram List:**
   - The original word is then appended to the list corresponding to its sorted signature in the `anagram_groups` dictionary.
7. **Result Creation:**
   - After processing all words, the algorithm creates a list of lists by extracting the values from the `anagram_groups` dictionary. These lists represent grouped anagrams.
8. **Return:**
   - The final result, a list of grouped anagrams, is returned.

Complexity

- Time complexity: O(N * K * log(K))
- Space complexity: O(N * K)

Code

```
class Solution:
    def groupAnagrams(self, strs: List[str]) -> List[List[str]]:
        anagram_groups = {}
        if len(strs) == 0:
            return strs
        else:
            for word in strs:
                sorted_word = ''.join(sorted(word))
                if sorted_word not in anagram_groups:
                    anagram_groups[sorted_word] = []
                anagram_groups[sorted_word].append(word)
            result = list(anagram_groups.values())
            return result
```

If you want to see more solutions to coding problems, you can visit:


GitHub