Group Anagrams

Difficulty	Medium
Category	Arrays
Question	https://leetcode.com/problems/group-anagrams/
Solution	https://youtu.be/vzdNOK2oB2E
Status	Done

Question

> Given an array of strings `strs`, group **the anagrams** together. You can return the answer in **any order**.
>
> An **Anagram** is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

Example

Example 1:

```
Input: strs = ["eat","tea","tan","ate","nat","bat"]
Output: [["bat"],["nat","tan"],["ate","eat","tea"]]
```

Example 2:

```
Input: strs = [""]
Output: [[""]]
```

Example 3:

```
Input: strs = ["a"]
Output: [["a"]]
```

Idea

Use array to record each string’s char frequency, and make it a common key. Add to this key in the table if there’s a match. O(n) complexity

Solution

```
class Solution:
    def groupAnagrams(self, strs: List[str]) -> List[List[str]]:
        anagrams = {}

        for word in strs:
            #Count the frequency of each character in the word
            char_count = [0]*26 # a....z

            for char in word:
                char_count[ord(char) - ord('a')] += 1
                # This line increments the count of the character in the char_count list at the index

            # Use a tuple of character counts as the key
            key = tuple(char_count)

            # If the key doesn't exist in the dictionary, create it
            if key not in anagrams:
                anagrams[key] = [word]

            else:
                anagrams[key].append(word)


        # Convert the values (lists of anagrams) from the dictionary to a list of lists
        return list(anagrams.values())
```

📌 **EXPLANATION**

1. n this approach, we use a list `char_count` of 26 zeros (assuming lowercase English letters). For each word, we count the frequency of each character in the word by incrementing the corresponding index in `char_count`.
2. We use a tuple `key` to represent the character counts. This tuple becomes the key in the dictionary.
3. If the `key` doesn't exist in the dictionary, we create a new entry with the `key` as the key and a list containing the current `word` as the value.
4. If the `key` already exists in the dictionary, we append the current `word` to the existing list of anagrams.
5. After processing all words, we convert the values from the `anagrams` dictionary (lists of anagrams) to a list of lists and return it as the result.

This updated approach has a time complexity of O(n * k) and is more efficient than the sorting-based approach, especially when dealing with longer words or a large number of words.