49. Group Anagrams

Medium Array Hash Table String Sorting

Leetcode Link

Problem Description

The task is to write a function that takes an array (list) of strings called `strs` and groups the anagrams together. An anagram is a word obtained by rearranging the letters of another word to form a new word using all the original characters exactly once. For example, "listen" and "silent" are anagrams of each other. The function can return the groups in any order, meaning the sequence of the groups is not important, and within each group, the sequence of strings is not important either.

Intuition

The intuition behind the solution is that if two strings are anagrams of each other, when sorted alphabetically, they will result in the same string. For instance, 'ate' and 'eat' when sorted will both result in 'aet'. This provides us with a simple way to identify groups of anagrams: by using the sorted string as a key in a dictionary (or hash map).

We can iterate through each string in our input list, sort the string alphabetically, and use the sorted string as a key in a dictionary. The value for each key will be a list that contains all the strings from the input that, when sorted, match this key. We use a defaultdict for convenience because it allows us to append to the list without needing to check if the key already exists -- if it doesn't, defaultdict automatically creates a new list by default.

At the end of the iteration, all anagrams will be grouped together in the dictionary under the same key, and we can then return the values of the dictionary, which are our required anagram groups.

Solution Approach

The solution provided uses a **hash table** (or dictionary in Python) to group the strings that are anagrams of each other. To implement this approach, the following steps are taken during the execution of the `groupAnagrams` function:

1. A `defaultdict` is created where each value is initialized as a list. This allows appending new items to the list without first checking if the key exists.
2. The function iterates through each string `s` in the provided list `strs`.
3. Inside the loop, the string `s` is sorted alphabetically using `sorted(s)` which returns a list of characters, and then `''.join(sorted(s))` is used to convert the list of characters back into a string. This sorted string serves as the key in our dictionary.
4. The original unsorted string `s` is appended to the list in the dictionary that corresponds to the sorted key (`k`). This means all anagrams of `s` will be grouped under the same key.
5. After processing all strings in `strs`, the function returns the values of the dictionary as a list of lists, where each sublist contains all the anagrams of a particular word from the input list.

Here is the relevant part of the code that demonstrates the algorithm:

```
d = defaultdict(list)
for s in strs:
    k = "".join(sorted(s))
    d[k].append(s)
return list(d.values())
```

Each key in the dictionary (`k`) represents a group of anagrams, and the corresponding value is a list containing all the strings that are anagrams of one another. Since the `defaultdict` automatically initializes missing keys with an empty list, no additional checks are necessary before appending to the list for a particular key. This makes the code clean and efficient.

The `groupAnagrams` function effectively utilizes sorting and hash tables to categorize strings by their anagram groups, ensuring that the full process is clear, intuitive and efficient, making it a standard approach to solve anagram grouping problems.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach:

Suppose we have the following array of strings: `["bat", "tab", "eat", "tea", "tan", "nat"]`.

Here is how we would group the anagrams together using the provided solution approach:

1. Create a `defaultdict` of lists: `d = defaultdict(list)`. This will store our anagrams.
2. Iterate through each string in the array:
   - `"bat"`: When sorted, becomes `"abt"`. We append `"bat"` to `d["abt"]`.
   - `"tab"`: When sorted, becomes `"abt"`. We append `"tab"` to `d["abt"]`.
   - `"eat"`: When sorted, becomes `"aet"`. We append `"eat"` to `d["aet"]`.
   - `"tea"`: When sorted, becomes `"aet"`. We append `"tea"` to `d["aet"]`.
   - `"tan"`: When sorted, becomes `"ant"`. We append `"tan"` to `d["ant"]`.
   - `"nat"`: When sorted, becomes `"ant"`. We append `"nat"` to `d["ant"]`.
3. Now, `d` looks like this:

   ```
   {
     "abt": ["bat", "tab"],
     "aet": ["eat", "tea"],
     "ant": ["tan", "nat"]
   }
   ```

   Each key corresponds to a sorted string, and the values are the original strings that are anagrams.
4. Finally, we return the dictionary values as a list of lists:

   ```
   [
     ["bat", "tab"],
     ["eat", "tea"],
     ["tan", "nat"]
   ]
   ```

   Each sublist contains anagrams of each other.

The order of the sublists and the order of the strings within each sublist does not matter. Those are our anagram groups, as requested. The function provides a clear and efficient way of splitting our input array into these anagram groups using sorting and hashing.

Python Solution

```
from collections import defaultdict

class Solution:
    def groupAnagrams(self, strs):
        # A dictionary that automatically creates a new list entry for each new key
        anagrams = defaultdict(list)

        # Iterate through each string in the provided list
        for s in strs:
            # Sort the string to create a key for the anagrams
            # Since sorted returns a list of characters, join them to form a string
            key = "".join(sorted(s))

            # Append the original string to the list of anagrams with the same key
            anagrams[key].append(s)

        # Convert the dictionary values to a list and return it
        return list(anagrams.values())

```

Java Solution

```
import java.util.*;

class Solution {
    public List<List<String>> groupAnagrams(String[] strs) {
        // Create a map to group the anagrams, where the key is the sorted string, and the value is a list of original strings.
        Map<String, List<String>> anagramsMap = new HashMap<>();

        // Iterate over each string in the array.
        for (String str : strs) {
            // Convert the string to a character array and sort it.
            char[] charArray = str.toCharArray();
            Arrays.sort(charArray);

            // Create a new string from the sorted character array.
            String sortedStr = String.valueOf(charArray);

            // If the sorted string key is not present in the map, initialize the list.
            // Then add the original string to the list associated with the sorted string key.
            anagramsMap.computeIfAbsent(sortedStr, key -> new ArrayList<>()).add(str);
        }

        // Return a new list containing all values from the map's lists,
        // effectively grouping all anagrams together.
        return new ArrayList<>(anagramsMap.values());
    }
}

```

C++ Solution

```
#include <vector>
#include <string>
#include <unordered_map>
#include <algorithm> // needed for std::sort()

class Solution {
public:
    // This function takes a list of strings and groups anagrams together.
    vector<vector<string>> groupAnagrams(vector<string>& strs) {
        // Create a map to keep track of the groups of anagrams.
        unordered_map<string, vector<string>> anagramGroups;

        // Iterate over each string in the input list.
        for (auto& str : strs) {
            // Create a key by sorting the characters in the string.
            string key = str; // Copy the original string to "key".
            sort(key.begin(), key.end()); // Sort the characters to form the key.

            // Add the original string to the vector corresponding to the sorted key.
            anagramGroups[key].emplace_back(str);
        }

        // Initialize a vector to hold the final grouped anagrams.
        vector<vector<string>> groupedAnagrams;

        // Iterate over the map to extract the groups of anagrams.
        for (auto& pair : anagramGroups) {
            // The second element of the pair is our group of anagrams.
            // Add this group to our final result.
            groupedAnagrams.emplace_back(pair.second);
        }

        // Return the grouped anagrams.
        return groupedAnagrams;
    }
};

```

Typescript Solution

```
// Function to group anagrams in an array of strings.
// Each group contains strings that are anagrams of each other.
function groupAnagrams(strs: string[]): string[][] {
    // Define a map to hold the sorted string as key and an array of its anagrams as value.
    const anagramMap: Map<string, string[]> = new Map();

    // Iterate through each string in the input array.
    for (const str of strs) {
        // Sort the characters of the string to form the key.
        const key = str.split('').sort().join('');

        // If the key is not present in the map, initialize it with an empty array.
        if (!anagramMap.has(key)) {
            anagramMap.set(key, []);
        }

        // Add the original string to the corresponding key's list of anagrams.
        anagramMap.get(key).push(str);
    }

    // Convert the values of the map to an array and return.
    return Array.from(anagramMap.values());
}

```

Time and Space Complexity

Time complexity

The time complexity of the code is primarily determined by two operations: the sorting operation for each string and the insertion into the dictionary.

1. **Sorting Each String**: Assuming the average length of the strings is `n` and there are `m` strings to sort, the time complexity of sorting each string using the TimSort algorithm (Python's default sort algorithm) is `O(n log n)`. Since we need to sort `m` such strings, the total time complexity for this part is `O(m * n log n)`.
2. **Inserting into Dictionary**: The insertion of `m` strings into the dictionary has a time complexity of `O(m)`, since each insertion operation into a dictionary is `O(1)` on average.

Combining these two, we get the total time complexity as `O(m * n log n)`.

Space Complexity

The space complexity is due to the storage of `m` entries in the dictionary and the lists of anagrams.

1. **Dictionary Storage (HashMap)**: The dictionary stores lists of anagrams. In the worst case, all strings are anagrams of each other, requiring `O(m * n)` space (since each of the `m` strings of length `n` can be in the same list).
2. **Sorted String Keys**: Additionally, for each string, we create a sorted copy to use as a key. Since we sort each string in place and only have `m` keys at any point in time, the space for the keys would be `O(m * n)`.

Therefore, the total space complexity of the algorithm is `O(m * n)`, as it dominates over the constant factors and smaller terms.