Solution: Group Anagrams

Let's solve the Group Anagrams problem using the Knowing What to Track pattern.

We'll cover the following

Statement

Given a list of words or phrases, group the words that are anagrams of each other. An anagram is a word or phrase formed from another word by rearranging its letters.

Constraints:

Let `strs` be the list of strings given as input to find the anagrams.

- $1 \leq$ `strs.length` $\leq 10^4$
- $0 \leq$ `strs[i].length` ≤ 100
- `strs[i]` consists of lowercase English letters.

> **Note:** The order in which the output is displayed doesn't matter.

Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

Naive approach

The naive approach is to sort all strings first, and then compare them to check whether they are identical. The idea is that if two strings are anagrams, sorting them both will always make them equal to each other. For example, consider the strings "car" and "arc". When sorted, both these strings become "acr".

We'll initialize a hash map to store the anagrams where the key represents the sorted string and the value represents the array of anagrams corresponding to that key. We'll run a loop on the given list of strings. On each iteration, we will sort the current string. We'll then check if the sorted string is present as a key in the hash map. If it is, we'll append the original, unsorted string to the array corresponding to that key. Otherwise, we'll add the new key-value pair to the hash map. At the end of the traversal, the hash map will contain all the groups of anagrams.

We'll need $O(n)$ time to traverse the list of strings of length n. In addition, we'll need $O(l \log(l))$ time to sort each string, where l represents the string with the greatest length in the array. So, the overall time complexity of the algorithm becomes $O(nl \log(l))$. The space complexity is $O(1)$, since the space used in the computation is also required in the output.

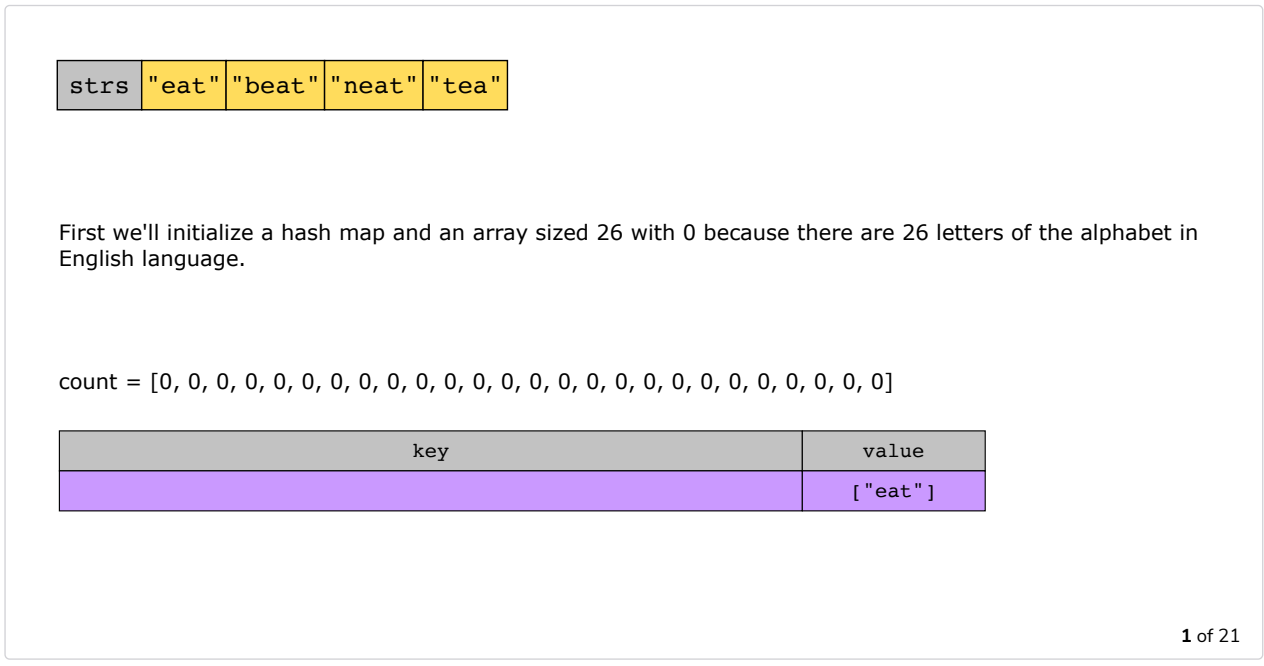
Optimized approach using frequency mapping

A better approach than sorting can be used to solve this problem. This solution involves computing the frequency of each letter in every string. This will help reduce the time complexity of the given problem. We'll just compute the frequency of every string and store the strings in their respective list in a hash map.

We see that all members of each set are characterized by the same frequency of each letter. This means that the frequency of each letter in the words belonging to the same group is equal. In the set `[["speed", "spede"]]`, the frequency of the characters `s`, `p`, `e`, and `d` are the same in each word.

Let's see how we can implement the above algorithm:

1. For each string, compute a 26-element list. Each element in this list represents the frequency of an English letter in the corresponding string. This frequency count will be represented as a tuple. For example, `"abbccc"` will be represented as `(1, 2, 3, 0, 0, ..., 0)`. This mapping will generate identical lists for strings that are anagrams.
2. Use this list as a key to insert the strings into a hash map. Between the counts, we'll insert "`#`"s such that it will look like `("#1#0#1#0#1#0#0#0#1......")`. All anagrams will be mapped to the same key in this hash map.
3. While traversing each string, we generate its 26-element list and check if this list is present as a key in the hash map. If it does, we'll append the string to the array corresponding to that key. Otherwise, we'll add the new key-value pair to the hash map.
4. Return the values of the hash map in a two-dimensional array, since each value will be an individual set of anagrams.



Let's look at the coded solution below:

Java

```java
class GroupAnagrams {
    public static List<List<String>> groupAnagrams(String[] strs){
      if (strs.length == 0)
            return new ArrayList<List<String>>();
        // Hashmap for count
        Map<String, List<String>> res = new HashMap<String, List<String>>();

        int[] count = new int[26]; // A place for every single alphabet in our string
        // We will compute the frequency for every string
        for (String s : strs) {
            Arrays.fill(count, 0);
            for (char c : s.toCharArray()){
                // Calculating the value from 1 to 26 for the alphabet
                int index = c - 'a';
                count[index]++;
            }

            StringBuilder delimStr = new StringBuilder("");
            for (int i = 0; i < 26; i++) {
                delimStr.append('#');
                delimStr.append(count[i]);
            }

            String key = delimStr.toString();
            if (!res.containsKey(key))
                res.put(key, new ArrayList<String>());
            // We add the string as an anagram if it matched the content
            // of our res hashmap
```

Group Anagrams

Solution summary

To recap, the solution to this problem can be divided into the following steps:

respectively. The key will be a character list of length 26, initialized to all 0s, and the value will be an array of anagrams.

2. Traverse over the list of strings and, for each string in the list, count the occurrence of its characters, and insert them as keys. Insert the string itself as a value in the hash map.
3. Use the resulting tuple of frequency as a key to check the occurrence of its anagram in the hash map. If the key already exists in the hash map, append it to the corresponding value. Otherwise, generate a new key-value pair to store the current string.
4. Repeat the above steps until all strings have been traversed and return the resultant list.

Time complexity

We count each letter for every string in a list, which results in a time complexity of $O(n)$, where n represents the size of the list. The overall time complexity will be $O(n \times k)$, where k denotes the size of the maximum length of a string in the list.

Space complexity

We store each string as a value in the dictionary whose size can be n, and the maximum size of the string can be k, the space complexity will be $O(n \times k)$.

Mark as Completed