# 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 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"]]

**Constraints:**

* 1 <= strs.length <= 104
* 0 <= strs[i].length <= 100
* strs[i] consists of lower-case English letters.

# Solution

#### **Approach 1: Categorize by Sorted String**

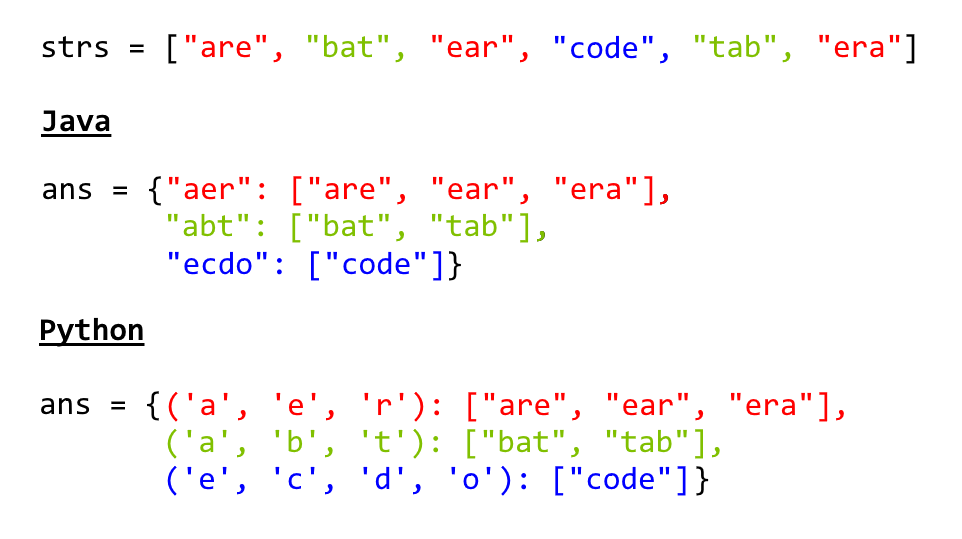
**Intuition**

Two strings are anagrams if and only if their sorted strings are equal.

**Algorithm**

Maintain a map ans : {String -> List} where each key \text{K}K is a sorted string, and each value is the list of strings from the initial input that when sorted, are equal to \text{K}K.

In Java, we will store the key as a string, eg. code. In Python, we will store the key as a hashable tuple, eg. ('c', 'o', 'd', 'e').



|  |
| --- |
| class Solution {  public List<List<String>> groupAnagrams(String[] strs) {  if (strs.length == 0) return new ArrayList();  Map<String, List> ans = new HashMap<String, List>();  for (String s : strs) {  char[] ca = s.toCharArray();  Arrays.sort(ca);  String key = String.valueOf(ca);  if (!ans.containsKey(key)) ans.put(key, new ArrayList());  ans.get(key).add(s);  }  return new ArrayList(ans.values());  }  } |

**Complexity Analysis**

* Time Complexity: O(NK \log K)*O*(*NK*log*K*), where N*N* is the length of strs, and K*K* is the maximum length of a string in strs. The outer loop has complexity O(N)*O*(*N*) as we iterate through each string. Then, we sort each string in O(K \log K)*O*(*K*log*K*) time.
* Space Complexity: O(NK)*O*(*NK*), the total information content stored in ans.

#### **Approach 2: Categorize by Count**

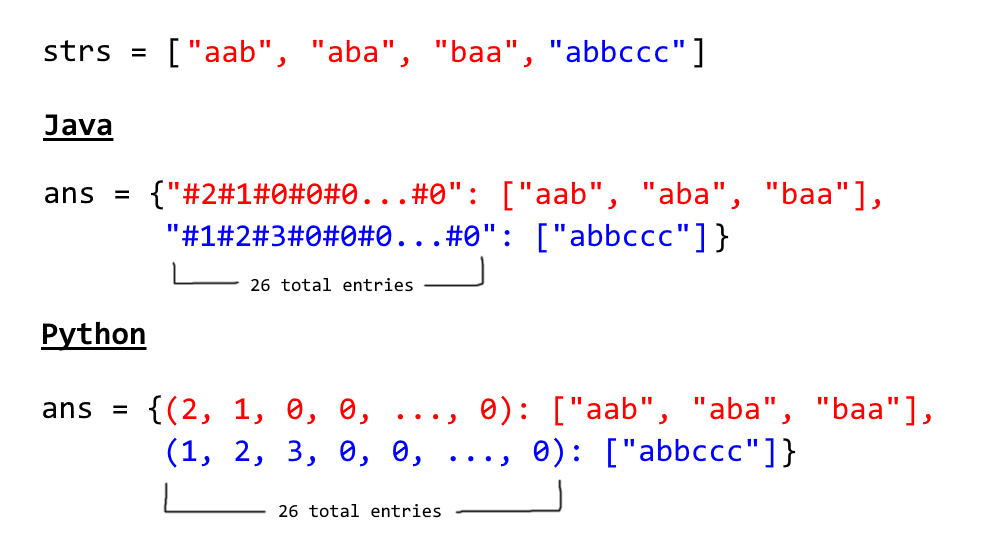
**Intuition**

Two strings are anagrams if and only if their character counts (respective number of occurrences of each character) are the same.

**Algorithm**

We can transform each string \text{s}s into a character count, \text{count}count, consisting of 26 non-negative integers representing the number of \text{a}a's, \text{b}b's, \text{c}c's, etc. We use these counts as the basis for our hash map.

In Java, the hashable representation of our count will be a string delimited with '**#**' characters. For example, abbccc will be #1#2#3#0#0#0...#0 where there are 26 entries total. In python, the representation will be a tuple of the counts. For example, abbccc will be (1, 2, 3, 0, 0, ..., 0), where again there are 26 entries total.



|  |
| --- |
| class Solution {  public List<List<String>> groupAnagrams(String[] strs) {  if (strs.length == 0) return new ArrayList();  Map<String, List> ans = new HashMap<String, List>();  int[] count = new int[26];  for (String s : strs) {  Arrays.fill(count, 0);  for (char c : s.toCharArray()) count[c - 'a']++;  StringBuilder sb = new StringBuilder("");  for (int i = 0; i < 26; i++) {  sb.append('#');  sb.append(count[i]);  }  String key = sb.toString();  if (!ans.containsKey(key)) ans.put(key, new ArrayList());  ans.get(key).add(s);  }  return new ArrayList(ans.values());  }  } |

Python

|  |
| --- |
| class Solution:  def groupAnagrams(strs):  ans = collections.defaultdict(list)  for s in strs:  count = [0] \* 26  for c in s:  count[ord(c) - ord('a')] += 1  ans[tuple(count)].append(s)  return ans.values() |

**Complexity Analysis**

* Time Complexity: O(NK)*O*(*NK*), where N*N* is the length of strs, and K*K* is the maximum length of a string in strs. Counting each string is linear in the size of the string, and we count every string.
* Space Complexity: O(NK)*O*(*NK*), the total information content stored in ans.