LeetCode 49 — Group Anagrams

1. Problem Title & Link

- **Title:** LeetCode 49 — Group Anagrams
- **Link:** https://leetcode.com/problems/group-anagrams/

2. Problem Statement (Short Summary)

Given an array of strings strs, group the strings that are **anagrams** of each other.
Return the groups in any order.

An *anagram* is a word formed by rearranging the letters of another (same letters, same counts).

Example: "eat", "tea", "ate" are anagrams and should be grouped together.

3. Examples (Input → Output)

Example 1

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

Output: [["eat","tea","ate"],["tan","nat"],["bat"]]

Example 2

Input: strs = [""]

Output: [[""]]

Example 3

Input: strs = ["a"]

Output: [["a"]]

4. Constraints

- 1 <= strs.length <= 10^4
- 0 <= strs[i].length <= 100
- strs[i] consists of lower-case English letters (typically).
- Output order of groups and of strings inside groups does not matter.

5. Core Concept (Pattern / Topic)

Hashing / Bucket by Signature

Map each string to a canonical signature that is identical for all its anagrams. Use that signature as a key in a hashmap to collect groups.

Common signatures:

- Sorted string (e.g., "eat" → "aet")
- Letter-count tuple (e.g., "eat" → counts of 26 letters)

This is a classic **hashmap grouping by key** pattern.

6. Thought Process (Step-by-Step Explanation)

Brute force idea (bad)

Compare every pair to test anagramness → O(n^2 * m) (m = avg len). Too slow.

Optimized idea (good)

For each string:

1. Compute a **signature** that is identical for all its anagrams.
    - Option A: sort the string → O(m log m)
    - Option B: build a 26-length count array → O(m) (better for long strings)
2. Use a hashmap key -> list of strings and append the original string to map[signature].
3. At the end, return all the lists (values of the map).

Choice guidance:

- If m small, sorting is concise and fine.
- If you want guaranteed O(m) per string, use count-key method (good when many long strings).

7. Visual / Intuition Diagram

strs = ["eat","tea","tan","ate","nat","bat"]

signatures (sorted):

"eat" -> "aet"

"tea" -> "aet"

"tan" -> "ant"

"ate" -> "aet"

"nat" -> "ant"

"bat" -> "abt"

map:

"aet" -> ["eat","tea","ate"]

"ant" -> ["tan","nat"]

"abt" -> ["bat"]

8. Pseudocode

```
map = {}
for s in strs:
    key = signature(s)    # either ''.join(sorted(s)) or counts tuple
    if key not in map:
        map[key] = []
    map[key].append(s)
```

```
return list(map.values())
```

9. Code Implementation

✅ Python (two variants: sorted-key and count-key)

Variant A — Sorted-key (simple)

```
from collections import defaultdict
from typing import List

class Solution:
    def groupAnagrams(self, strs: List[str]) -> List[List[str]]:
        d = defaultdict(list)
        for s in strs:
            key = ''.join(sorted(s))    # O(m log m)
            d[key].append(s)
        return list(d.values())
```

Variant B — Count-key (O(m) per string)

```
from collections import defaultdict
from typing import List, Tuple

class Solution:
    def groupAnagrams(self, strs: List[str]) -> List[List[str]]:
        d = defaultdict(list)
        for s in strs:
            count = [0] * 26
            for ch in s:
                count[ord(ch) - ord('a')] += 1
            key = tuple(count)    # immutable key
            d[key].append(s)
        return list(d.values())
```

✅ Java (sorted-key and count-key)

Variant A — Sorted-key

```
class Solution {
    public List<List<String>> groupAnagrams(String[] strs) {
        Map<String, List<String>> map = new HashMap<>();
        for (String s : strs) {
            char[] arr = s.toCharArray();
            Arrays.sort(arr);
            String key = new String(arr);
            map.computeIfAbsent(key, k -> new ArrayList<>()).add(s);
        }
        return new ArrayList<>(map.values());
    }
}
```

Variant B — Count-key (faster for long strings)

```
class Solution {
    public List<List<String>> groupAnagrams(String[] strs) {
        Map<String, List<String>> map = new HashMap<>();
        for (String s : strs) {
            int[] count = new int[26];
            for (char c : s.toCharArray()) count[c - 'a']++;
            // build key like "1#0#0#2#..."; using StringBuilder
            StringBuilder sb = new StringBuilder();
            for (int i = 0; i < 26; i++) {
                sb.append('#');
                sb.append(count[i]);
            }
            String key = sb.toString();
            map.computeIfAbsent(key, k -> new ArrayList<>()).add(s);
        }
        return new ArrayList<>(map.values());
    }
}
```

10. Time & Space Complexity

Using sorted-key:

- Let n = number of strings, m = max length of a string.
- **Time:** O(n * m log m) (sorting each string)
- **Space:** O(n * m) for hashmap and keys

Using count-key:

- **Time:** O(n * m) (counting letters for each string)
- **Space:** O(n * m) (store groups + keys; count-key uses O(26) per key though)

Both return total grouped strings; order doesn’t matter.

11. Common Mistakes / Edge Cases

- Using a mutable list as a dict key (must be immutable: use tuple or string).
- Not handling empty string "" properly (signature of "" should be empty key — works).
- Forgetting to convert count array to tuple/string when using as key.
- Using expensive string concatenation inside loops in some languages (use StringBuilder in Java).
- Assuming output order; tests accept any order of groups and group members.

Edge cases:

- [""] → groups: [[""]]
- large number of short strings vs few very long strings (choose algorithm accordingly)

12. Detailed Dry Run (Step-by-Step Table)

Input:

["eat","tea","tan","ate","nat","bat"]

We'll use **sorted-key** for dry run.

Step	s	sorted(s)	map after step
1	"eat"	"aet"	{"aet": ["eat"]}
2	"tea"	"aet"	{"aet": ["eat","tea"]}
3	"tan"	"ant"	{"aet": ["eat","tea"], "ant": ["tan"]}
4	"ate"	"aet"	{"aet": ["eat","tea","ate"], "ant": ["tan"]}
5	"nat"	"ant"	{"aet": ["eat","tea","ate"], "ant": ["tan","nat"]}
6	"bat"	"abt"	{"aet": ["eat","tea","ate"], "ant": ["tan","nat"], "abt": ["bat"]}

Return values:

[["eat","tea","ate"], ["tan","nat"], ["bat"]]

(Order of groups may vary.)

13. Common Use Cases (Real-Life / Interview)

- Grouping words by similarity (anagrams) — e.g., dictionary clustering
- Search optimization: normalize queries to canonical form
- Detecting permutations of patterns in text
- Data normalization for hashing/grouping

14. Common Traps

- Using sorting inside heavy loops for extremely large input (use count-key when needed).
- Building string keys inefficiently (in Java, prefer StringBuilder).
- Forgetting that keys must uniquely represent all anagrams (e.g., collisions).

15. Builds To (Related LeetCode Problems)

- **LC 242** — Valid Anagram (single pair check)
- **LC 438** — Find All Anagrams in a String (sliding window + freq)
- **LC 726** — Count of Anagrams (variation)

- **LC 187** — Repeated DNA Sequences (hash + sliding window with fixed-length)

16. Alternate Approaches + Comparison

Approach	Time	Space	Pros / Cons
Sorted-key	O(n m log m)	O(n m)	Simple, concise; fine when m small
Count-key	O(n m)	O(n m)	Faster for long strings; slightly more code
Pairwise comparisons	O(n^2 m)	O(1)	Too slow for constraints

17. Why This Solution Works (Short Intuition)

Anagrams have identical character multiset. By converting each string into a canonical signature (sorted chars or letter counts), all anagrams map to the same key; grouping by that key collects all anagrams efficiently.

18. Variations / Follow-Up Questions

- What if strings contain uppercase or Unicode characters? (Adjust count or sort accordingly.)
- How to return groups in deterministic order? (Sort groups and/or sort list of groups.)
- How to find top-k largest anagram groups?
- Streaming version: group anagrams from a stream of words (use incremental hashing + external storage).