

Group Anagrams

# Attempts	4
📅 Date Solved	@October 23, 2025
🕒 Difficulty	Medium
📅 Next Review	@October 28, 2025
🕒 Status	Solved
☰ Topic/Pattern	Array and String

Link → <https://neetcode.io/problems/anagram-groups?list=neetcode150>

Problem

- Given an array of strings `strs`, group all the **anagrams** together into sublists.
- Return the grouped anagrams in any order.

Example

Input	Output	Reason
<code>["act","pots","tops","cat","stop","hat"]</code>	<code>[['act','cat'], ['pots','tops','stop'], ['hat']]</code>	'act' & 'cat' share same letters; 'pots','tops','stop' too
<code>["abc","bca","cab"]</code>	<code>[['abc','bca','cab']]</code>	all are anagrams
<code>["a","b","c"]</code>	<code>[['a'], ['b'], ['c']]</code>	all unique

Approaches

1. Brute Force — Compare each pair

```

def group_anagrams(strs):
    visited = [False] * len(strs)
    result = []

    for i in range(len(strs)):
        if not visited[i]:
            group = [strs[i]]
            visited[i] = True

            for j in range(i + 1, len(strs)):
                if sorted(strs[i]) == sorted(strs[j]):
                    group.append(strs[j])
                    visited[j] = True

            result.append(group)

    return result

```

- **Time:** $O(n^2 \times k \log k)$ — Compare every pair and sort each word
- **Space:** $O(n)$ — For tracking visited words
- **Notes:** Works but very slow for large input sizes

2. Sorting Key — Use sorted word as a key

```

from collections import defaultdict

def group_anagrams(strs):
    anagrams = defaultdict(list)
    #defaultdict(<class 'list'>, {})
    for word in strs:
        sorted_word = sorted(word)
        #['a', 'e', 't']
        key = ''.join(sorted_word)

```

```
#aet
anagrams[key].append(word)
#defaultdict(<class 'list'>, {'aet': ['eat']})

return list(anagrams.values())
```

- **Time:** $O(n \times k \log k)$ — Sort each word once
 - **Space:** $O(n \times k)$ — For dictionary and grouped words
 - **Notes:** Clean, simple, widely accepted solution
-

3. Frequency Count Key (Best) — Use character frequency tuple

```
from collections import defaultdict

def group_anagrams(strs):
    anagrams = defaultdict(list)

    for word in strs:
        count = [0] * 26
        for c in word:
            count[ord(c) - ord('a')] += 1
        anagrams[tuple(count)].append(word)

    return list(anagrams.values())
```

- **Time:** $O(n \times k)$ — Count characters directly (no sorting)
 - **Space:** $O(n \times k)$ — Store frequency tuples as keys
 - **Notes:** Most efficient; scales well for large inputs
-

Summary

Approach	Time	Space	Notes
Brute Force	$O(n^2 \times k \log k)$	$O(n)$	Slow, not practical
Sorting Key	$O(n \times k \log k)$	$O(n \times k)$	Clean and easy
Frequency Count	$O(n \times k)$	$O(n \times k)$	Fastest and most optimal

Edge Cases

Input	Output	Reason
<code>[]</code>	<code>[]</code>	Empty input
<code>["a"]</code>	<code>[["a"]]</code>	Single element
<code>["", ""]</code>	<code>[["", ""]]</code>	Empty strings are anagrams
<code>["ab", "ba", "abc"]</code>	<code>[["ab", "ba"], ["abc"]]</code>	Only first two match

Tip

- Always remember: sorting is intuitive but costly for longer strings.
- For high efficiency, prefer **character frequency** keys since they avoid sorting altogether.
- This problem tests **hashing**, **string manipulation**, and **algorithmic optimization** skills.