Group Anagrams - Study Guide

Problem Statement

Given an array of strings, group all anagrams together. Return the grouped anagrams in any order.

Example:

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

Key Insights

What Makes This Problem Interesting

1. **Pattern Recognition**: Anagrams share the same characters in different orders
2. **Key Generation**: Need a way to identify anagrams using a common identifier
3. **Grouping Strategy**: Multiple strings can belong to the same group

The Core Breakthrough

Sorted characters create a unique signature for anagrams

- "eat" → "aet"
- "tea" → "aet"
- "ate" → "aet"

All three map to the same sorted key!

Solution Approach

Algorithm Overview

1. Create a HashMap to store groups
2. For each string:

    - Sort its characters to create a key
    - Add the original string to the group for that key
3. Return all grouped values

Implementation Details

```
function groupAnagrams(strs: string[]): string[][] {
  const hashMap = new Map();

  strs.forEach((word) => {
    // Create sorted key
    const sortedWord = word.split("").sort().join("");

    // Group by sorted key
    if (hashMap.has(sortedWord)) {
      hashMap.get(sortedWord).push(word);
    } else {
      hashMap.set(sortedWord, [word]);
    }
  });

  return [...hashMap.values()];
}
```

Complexity Analysis

Time Complexity: O(n × k log k)

- **n** = number of strings
- **k** = maximum length of a string
- For each string: O(k log k) to sort the characters
- Total: O(n × k log k)

Space Complexity: O(n × k)

- HashMap stores all strings
- Keys are sorted versions of strings
- Total space proportional to input size

Common Mistakes to Avoid

1. Nested Loop Approach

Don't do this:

```
// Trying to compare every string with every other string
for (let i = 0; i < strs.length; i++) {
  for (let j = 0; j < strs.length; j++) {
    // Compare characters...
  }
}
```

Why it fails:

- O(n^2) complexity
- Difficult to handle grouping correctly
- Complex edge case handling
- Modifying array while iterating (splice) causes bugs

2. Character Comparison Issues

Problem: Using `every()` and `includes()` for character comparison

- Doesn't handle duplicate characters correctly
- "aab" vs "aba" - false positive matching

3. Index Management

- Trying to track indices while grouping
- Splicing arrays during iteration changes indices
- Leads to incorrect grouping

Edge Cases to Consider

Empty Strings

```
Input: [""]
Output: [[""]]

Input: ["", "", ""]
Output: [["", "", ""]]
```

- Empty string sorted is still empty string

- All empty strings group together

Single Character

```
Input: ["a"]
Output: [["a"]]
```

No Anagrams

```
Input: ["bat", "dog", "cat"]
Output: [["bat"], ["dog"], ["cat"]]
```

- Each word gets its own group

All Anagrams

```
Input: ["abc", "bca", "cab"]
Output: [["abc", "bca", "cab"]]
```

- All belong to one group

Alternative Approaches

Character Count as Key

Instead of sorting, use character frequency:

```
// Create key: "a1b1c1" for "abc"
const createKey = (str: string) => {
  const count = new Array(26).fill(0);
  for (let char of str) {
    count[char.charCodeAt(0) - 'a'.charCodeAt(0)]++;
  }
  return count.join('#');
};
```

Pros:

- O(k) key generation vs O(k log k) for sorting
- Better time complexity overall: O(n × k)

Cons:

- More complex implementation
- Only works for lowercase English letters

Problem-Solving Process

Initial Approach

1. ✅ Correct intuition: Use a HashMap
2. ❌ Wrong path: Try nested loops with splicing
3. ✅ Return to HashMap idea
4. ✅ Realize sorted characters work as keys

Key Learning

- **First instinct was right** - trust your initial algorithm design
- **Avoid over-complicating** - simple HashMap is sufficient
- **Don't modify while iterating** - causes index issues

Related Patterns

1. **Grouping with HashMap**
    - Two Sum
    - Longest Consecutive Sequence
2. **String Manipulation**
    - Valid Anagram
    - Find All Anagrams in String
3. **Character Sorting**
    - Reorganize String
    - Sort Characters by Frequency

Practice Questions

1. How would you modify this to return only groups with more than one anagram?
2. What if the strings can contain uppercase letters and special characters?
3. How would you optimize if all strings are the same length?
4. Can you implement the character count approach?

Key Takeaways

1. **HashMap for grouping** - When you need to group items by a common property
2. **Sorted string as key** - Simple and effective for anagram detection
3. **Avoid nested loops** - HashMap provides O(1) lookup
4. **Trust your instincts** - Initial algorithm ideas are often correct
5. **Edge cases matter** - Empty strings and duplicates need handling