Day 13

Prepared by Azan Imtiaz

Problem 1

Group Anagram

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"]]
```

Approach 1: Grouping Anagrams by Sorted Strings

In this approach, we group anagrams by sorting the characters in each string. Here's how it works:

- We iterate through each string in the input array.
- For each string, we convert it to a character array, sort the array, and then convert it back to a string. This sorted string serves as a key.

- We use a HashMap where the key is the sorted string, and the value is a list of strings that are anagrams of each other.
- If the sorted string is not already a key in the map, we add it with a new list as its value.
- We add the original string to the list corresponding to the sorted key.
- Finally, we return a collection of these lists.

Time Complexity: The time complexity for this approach is O(n * k log k), where n is the number of strings in the input array and k is the maximum length of a string. This is because we sort each string which takes O(k log k) time.

Here's the Java code for Approach 1:

```
public List<List<String>> groupAnagrams(String[] strs) {
    Map<String, List<String>> map = new HashMap<>();

    for (String str : strs) {
        char[] charArray = str.toCharArray();
        java.util.Arrays.sort(charArray);
        String sortedStr = new String(charArray);

        if (!map.containsKey(sortedStr)) {
            map.put(sortedStr, new ArrayList<>());
        }

        map.get(sortedStr).add(str);
    }

    return new ArrayList<>(map.values());
}
```

Dry Run Example for Approach 1:

Let's dry run this code with a small example: `strs = ["bat", "tab", "eat"]`.

- For "bat", after sorting we get "abt". Add "bat" to the list under key "abt".
- For "tab", after sorting we also get "abt". Add "tab" to the list under key "abt".
- For "eat", after sorting we get "aet". Add "eat" to the list under key "aet".

  The map now has two keys "abt" -> ["bat", "tab"] and "aet" -> ["eat"].

- Return the values of the map as a list of lists.

Approach 2: Grouping Anagrams by Character Frequency

This approach groups anagrams based on the frequency of each character in the strings. Here's the process:

- We create a frequency string for each input string, which consists of concatenated pairs of characters and their counts in the string.
- We use a HashMap where the key is the frequency string, and the value is a list of strings that have the same character frequencies.
- If the frequency string is already a key in the map, we add the original string to the corresponding list.
- If it's not, we create a new list and add the string to it.
- Finally, we return a collection of these lists.

Time Complexity: The time complexity for this approach is O(n * k), where n is the number of strings and k is the maximum length of a string. This is because creating a frequency string for each input string takes O(k) time.

Here's the Java code for Approach 2:

```
class Solution {
    public List<List<String>> groupAnagrams(String[] strs) {
        if (strs == null || strs.length == 0)
            return new ArrayList<>();

        Map<String, List<String>> frequencyStringsMap = new HashMap<>();
        for (String str : strs) {
            String frequencyString = getFrequencyString(str);

            if (!frequencyStringsMap.containsKey(frequencyString)) {
                frequencyStringsMap.put(frequencyString, new ArrayList<>());
            }
            frequencyStringsMap.get(frequencyString).add(str);
        }
        return new ArrayList<>(frequencyStringsMap.values());
    }
```

```
    private String getFrequencyString(String str) {
        int[] freq = new int[26];
        for (char c : str.toCharArray()) {
            freq[c - 'a']++;
        }

        StringBuilder frequencyString = new StringBuilder("");
        for (int i = 0; i < freq.length; i++) {
            if (freq[i] > 0) {
                frequencyString.append((char) (i + 'a'));
                frequencyString.append(freq[i]);
            }
        }
        return frequencyString.toString();
    }
}
```

Dry Run Example for Approach 2:

Using the same example: `strs = ["bat", "tab", "eat"]`.

- For "bat", the frequency string is "a1b1t1". Add "bat" to the list under key "a1b1t1".
- For "tab", the frequency string is also "a1b1t1". Add "tab" to the list under key "a1b1t1".
- For "eat", the frequency string is "a1e1t1". Add "eat" to the list under key "a1e1t1".
- The map now has two keys "a1b1t1" -> ["bat", "tab"] and "a1e1t1" -> ["eat"].
- Return the values of the map as a list of lists.

Problem 2

Longest Palindromic Substring

Given a string `s`, return *the longest palindromic substring* in `s`.

Example 1:

```
Input: s = "babad"
Output: "bab"
Explanation: "aba" is also a valid answer.
```

Example 2:

```
Input: s = "cbbd"
Output: "bb"
```

Intuition Behind the Solution

The intuition for finding the longest palindromic substring is to expand around each character in the string, treating it as the center of a potential palindrome. Since palindromes are symmetrical, we can check for palindromes by expanding outwards from the center and ensuring that the characters are the same on both sides.

Step-by-Step Approach

1. Initialize variables `start` and `end` to keep track of the beginning and end of the longest palindromic substring found so far.
2. Iterate through each character in the string with index `i`.
3. For each character, try to expand around the center in two ways:
   - Once considering the current character as the center (`len2`).
   - Once considering the space between the current character and the next as the center (`len1`), which accounts for even-length palindromes.
4. Calculate the maximum length of the palindrome from these two attempts.
5. If this length is greater than the length of the currently known longest palindrome, update `start` and `end` to the new values.
6. After the loop, extract the longest palindromic substring using `substring(start, end + 1)`.

Code

```
class Solution {

    public String longestPalindrome(String s) {
        int start = 0;
        int end = 0;
        for (int i = 0; i < s.length(); i++) {
            int len1 = expandFromCenter(s, i, i + 1);
            int len2 = expandFromCenter(s, i, i);
```

```
                int len = Math.max(len1, len2);
                if (end - start < len) {
                    start = i - (len - 1) / 2;
                    end = i + len / 2;
                }
            }
            return s.substring(start, end + 1);
        }

        private int expandFromCenter(String s, int left, int right) {
            while (left >= 0 && right < s.length() && s.charAt(left) == s.charAt(right)) {
                left--;
                right++;
            }
            return right - left - 1;
        }
    }
```

Time Complexity

The time complexity of this solution is O(n^2), where n is the length of the input string. This is because for each character, we potentially expand across the entire length of the string in the worst case.

Dry Run

Let's dry run the solution with an example string `"babad"`:

1. Start with `i = 0`, `start = 0`, `end = 0`.
2. At `i = 0`, `len1 = 0` (no even palindrome), `len2 = 1` ("b").
3. Update `start` and `end` to `0` and `0`.
4. Move to `i = 1`, `len1 = 2` ("ab"), `len2 = 1` ("a").
5. Update `start` and `end` to `0` and `1`.
6. Continue this process until all characters have been checked.
7. The longest palindrome found is "bab" with `start = 0` and `end = 2`.
8. Return the substring `"bab"`.