<https://leetcode.com/problems/find-and-replace-pattern/description/>

Given a list of strings words and a string pattern, return a list of words[i] that match pattern. You may return the answer in any order.

A word matches the pattern if there exists a permutation of letters p so that after replacing every letter x in the pattern with p(x), we get the desired word.

Recall that a permutation of letters is a bijection from letters to letters: every letter maps to another letter, and no two letters map to the same letter.

**Example 1:**

**Input:** words = ["abc","deq","mee","aqq","dkd","ccc"], pattern = "abb"

**Output:** ["mee","aqq"]

**Explanation:** "mee" matches the pattern because there is a permutation {a -> m, b -> e, ...}.

"ccc" does not match the pattern because {a -> c, b -> c, ...} is not a permutation, since a and b map to the same letter.

**Example 2:**

**Input:** words = ["a","b","c"], pattern = "a"

**Output:** ["a","b","c"]

**Constraints:**

1 <= pattern.length <= 20

1 <= words.length <= 50

words[i].length == pattern.length

pattern and words[i] are lowercase English letters.

**Attempt 1: 2025-03-24**

**Solution 1: Hash Table (30 min)**

**Style 1: Two Maps**

class Solution {

    public List<String> findAndReplacePattern(String[] words, String pattern) {

        List<String> result = new ArrayList<>();

        for(String word : words) {

            if(match(word, pattern)) {

                result.add(word);

            }

        }

        return result;

    }

    private boolean match(String word, String pattern) {

        char[] wToP = new char[26];

        char[] pToW = new char[26];

        for(int i = 0; i < word.length(); i++) {

            char wChar = word.charAt(i);

            char pChar = pattern.charAt(i);

            int wIdx = wChar - 'a';

            int pIdx = pChar - 'a';

            if(wToP[wIdx] == 0 && pToW[pIdx] == 0) {

                wToP[wIdx] = pChar;

                pToW[pIdx] = wChar;

            } else {

                if(wToP[wIdx] != pChar || pToW[pIdx] != wChar) {

                    return false;

                }

            }

        }

        return true;

    }

}

Time Complexity: O(N\*L), where N is the number of words and L is the length of each word.

Space Complexity: O(N\*L)

**Style 2: Single Map (Inspired by** [L290.Word Pattern (Ref.L205,L890)](note://F76ECAEBA42D4F90B5BC0D78CA816BA7)**)**

class Solution {

    public List<String> findAndReplacePattern(String[] words, String pattern) {

        List<String> result = new ArrayList<String>();

        for(String word : words) {

            if(match(word, pattern)) {

                result.add(word);

            }

        }

        return result;

    }

    private boolean match(String word, String pattern) {

        Map<Character, Character> map = new HashMap<>();

        for(int i = 0; i < word.length(); i++) {

            char w = word.charAt(i);

            char p = pattern.charAt(i);

            // If no current {w, p} mapping, the {w, p} mapping should be the first

            if(!map.containsKey(w)) {

                // But if already contains the projection value of this key,

                // which means the value match another key not relate as

                // one on one projection.

                // E.g word = "kd", pattern = "bb"

                // First putting ('k', 'b') on map, key = 'k' not in map now,

                // when we try to put ('d', 'b') on map, we find 'b' already

                // exist in map's value and is projection of 'k', it violates

                // one on one projection relation, so return false

                if(map.containsValue(p)) {

                    return false;

                }

                map.put(w, p);

            } else {

                // Check if the existing mapping matches the current pattern

                // character, which means if the mapping exists already, the

                // {w, p} should keep the same, projection cannot be other value

                if(map.get(w) != p) {

                    return false;

                }

            }

        }

        return true;

    }

}

Time Complexity: O(N\*L), where N is the number of words and L is the length of each word.

Space Complexity: O(N\*L)

**Solution 2: Normalize (30 min)**

class Solution {

    public List<String> findAndReplacePattern(String[] words, String pattern) {

        List<String> result = new ArrayList<>();

        String normalizedPattern = normalize(pattern);

        for(String word : words) {

            if(normalize(word).equals(normalizedPattern)) {

                result.add(word);

            }

        }

        return result;

    }

    private String normalize(String str) {

        Map<Character, Character> map = new HashMap<>();

        char nextChar = 'a';

        char[] chars = str.toCharArray();

        for(int i = 0; i < chars.length; i++) {

            char c = chars[i];

            if(!map.containsKey(c)) {

                map.put(c, nextChar);

                nextChar++;

            }

            chars[i] = map.get(c);

        }

        return new String(chars);

    }

}

Time Complexity: O(N\*L), where N is the number of words and L is the length of each word.

Space Complexity: O(N\*L)

**Refer to Deepseek**

To solve this problem, we need to determine which words from a given list match a specified pattern. A word matches the pattern if there exists a bijection between the characters of the pattern and the word. This means each character in the pattern must map to exactly one character in the word, and vice versa.

**Approach 1: Hash Table**

**Two Maps**

Check Lengths: First, ensure the word and the pattern have the same length. If not, the word cannot match the pattern.

Bijection Check: Use two arrays to track the mappings between characters of the pattern and the word. One array maps characters from the pattern to the word, and the other maps characters from the word to the pattern.

Iterate Through Characters: For each character in the pattern and the corresponding character in the word, check if they follow the established mappings. If a conflict is detected (i.e., a character in the pattern maps to a different character in the word or vice versa), the word does not match the pattern.

**Solution Code**

import java.util.ArrayList;

import java.util.List;

class Solution {

public List<String> findAndReplacePattern(String[] words, String pattern) {

List<String> result = new ArrayList<>();

for (String word : words) {

if (matches(word, pattern)) {

result.add(word);

}

}

return result;

}

private boolean matches(String word, String pattern) {

if (word.length() != pattern.length()) {

return false;

}

char[] pToW = new char[26];

char[] wToP = new char[26];

for (int i = 0; i < word.length(); i++) {

char pChar = pattern.charAt(i);

char wChar = word.charAt(i);

int pIdx = pChar - 'a';

int wIdx = wChar - 'a';

if (pToW[pIdx] == 0 && wToP[wIdx] == 0) {

pToW[pIdx] = wChar;

wToP[wIdx] = pChar;

} else {

if (pToW[pIdx] != wChar || wToP[wIdx] != pChar) {

return false;

}

}

}

return true;

}

}

**Explanation**

Bijection Check: The helper function matches uses two arrays, pToW and wToP, to track the mappings from pattern to word and word to pattern respectively. These arrays ensure that each character in the pattern maps to a unique character in the word and vice versa.

Efficiency: By using arrays of fixed size (26 for each lowercase letter), we ensure that the solution is efficient, with a time complexity of O(N \* M), where N is the number of words and M is the length of the longest word. This is efficient given the constraints of typical input sizes for such problems.

Edge Cases: The solution handles edge cases such as words of different lengths and ensures correct bijection checks for all characters.

This approach efficiently checks for the bijection requirement and ensures that only valid words are included in the result.

**Single Map**

Single Map with Bidirectional Check: Use a single map to track the mapping from word characters to pattern characters. Additionally, ensure that each pattern character is not mapped by multiple word characters.

Check Existing Mappings: When encountering a word character that's already mapped, verify it maps to the current pattern character. If not, the word doesn't match the pattern.

Prevent Multiple Mappings: When adding a new word character, ensure the corresponding pattern character isn't already mapped by another word character.

**Solution Code**

import java.util.ArrayList;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

class Solution {

public List<String> findAndReplacePattern(String[] words, String pattern) {

List<String> result = new ArrayList<>();

for (String word : words) {

if (match(word, pattern)) {

result.add(word);

}

}

return result;

}

private boolean match(String word, String pattern) {

if (word.length() != pattern.length()) {

return false;

}

Map<Character, Character> map = new HashMap<>();

for (int i = 0; i < word.length(); i++) {

char w = word.charAt(i);

char p = pattern.charAt(i);

if (!map.containsKey(w)) {

// Check if p is already mapped by another character

if (map.containsValue(p)) {

return false;

}

map.put(w, p);

} else {

// Check if the existing mapping matches the current pattern character

if (map.get(w) != p) {

return false;

}

}

}

return true;

}

}

**Explanation**

Single Map Usage: The map tracks each word character's corresponding pattern character.

Bidirectional Validation:

When a new word character is encountered, check if the pattern character is already mapped to another word character using containsValue.

For existing mappings, ensure the current pattern character matches the previously mapped character.

Efficiency: The solution efficiently checks each character in linear time, O(n), where n is the length of the words and pattern. The space complexity is O(1) due to the fixed-size character set.

This approach ensures that both the word and pattern maintain a strict bijection, correctly identifying all valid matches.

**Approach 2: Normalize**

To solve this problem, we need to determine which words from a given list match a specified pattern. A word matches the pattern if there exists a bijection between the characters of the pattern and the word. This means each character in the pattern must map to exactly one character in the word, and vice versa.

**Approach**

Normalization: Convert both the pattern and each word into a normalized form where each character is replaced by a sequential character starting from 'a' based on their first occurrence. This ensures that two strings with the same structure of character repetition will have the same normalized form.

Comparison: Compare the normalized form of each word with the normalized form of the pattern. If they match, the word is added to the result list.

**Solution Code**

class Solution {

public List<String> findAndReplacePattern(String[] words, String pattern) {

List<String> result = new ArrayList<>();

String normalizedPattern = normalize(pattern);

for (String word : words) {

if (normalize(word).equals(normalizedPattern)) {

result.add(word);

}

}

return result;

}

private String normalize(String s) {

HashMap<Character, Character> map = new HashMap<>();

char nextChar = 'a';

char[] arr = s.toCharArray();

for (int i = 0; i < arr.length; i++) {

char c = arr[i];

if (!map.containsKey(c)) {

map.put(c, nextChar);

nextChar++;

}

arr[i] = map.get(c);

}

return new String(arr);

}

}

**Explanation**

Normalization Function: The normalize function converts a string into its normalized form by mapping each unique character to the next sequential character starting from 'a'. This ensures that the structure of character repetition is preserved.

Comparison: The main function converts the pattern into its normalized form once. Each word is then normalized and compared to the normalized pattern. If they match, the word is added to the result list.

Efficiency: The normalization process runs in O(n) time for each word, where n is the length of the word. The overall time complexity is O(N \* M), where N is the number of words and M is the average length of the words, which is efficient given typical input constraints.

**Refer to**

Approach 1: Two Maps

Intuition and Algorithm

**If say, the first letter of the pattern is "a", and the first letter of the word is "x", then in the permutation, "a" must map to "x".**

We can write this bijection using two maps: a forward map m1 and a backwards map m2.

m1:"a"→"x"

m2:"x"→"a"

Then, if there is a contradiction later, we can catch it via one of the two maps. For example, if the (word, pattern) is ("aa", "xy"),

we will catch the mistake in m1("a")="x"="y". Similarly, with (word, pattern) = ("ab", "xx"), we will catch the mistake in m2.

**Refer to**

[L205.Isomorphic Strings (Ref.L290,L890)](note://WEB87f5bda440967963b486a4c5368709a7)

[L290.Word Pattern (Ref.L205,L890)](note://F76ECAEBA42D4F90B5BC0D78CA816BA7)