# 500. Keyboard Row

### Hash Table String

**Problem Description** 

In this problem, we are given an array of strings named words. Our task is to identify which of these words can be typed using letters from only one row on an American keyboard. The American keyboard has three rows of letters:

The second row contains the letters "asdfghjkl".

• The first row contains the letters "qwertyuiop".

The third row contains the letters "zxcvbnm".

Intuition

To solve this problem, we create a mapping that tells us which row each letter of the alphabet belongs to. We then iterate over each word in the given words array and for each word, we check if all its letters come from the same row.

A word qualifies if all of its letters are found in one of these rows. The goal is to return a list of words that meet this criterion.

alphabet and its value represents the row number on the keyboard ('1' for the first row, '2' for the second row, and so on). This allows us to quickly check the row of a letter by computing its relative position in the alphabet ('a' as index 0, 'b' as index 1, etc.) For instance, s [0] would give us the row number for the letter 'a', which is '1'.

To facilitate the row checking process, we create a string s where the position of each character corresponds to a letter of the

For each word, we start by looking up the row of its first letter. Then we use a simple loop (or a comprehension in Python) to check if every subsequent letter in the word is from the same row. If they all match the row of the first letter, we add the word to the answer list ans. After we've checked all words, we return the list ans of words that can be typed using only one row of the keyboard.

from one row of the keyboard and is returned.

step ensures comparison is case-insensitive.

second row. Therefore, "Hello" does not qualify.

the second row too. Thus, "Dad" qualifies as well.

letters from only one row on the keyboard.

row\_mapping = "12210111011122000010020202"

valid\_words.append(word)

public String[] findWords(String[] words) {

// List to store the eligible words

// Iterate over the array of words

boolean canBeTyped = true;

// Return the filtered list of words

function findWords(words: string[]): string[] {

const validWords: string[] = [];

let isWordValid = true;

break;

if (isWordValid) {

for (const word of words) {

const keyboardRowMapping = '12210111011122000010020202';

// Iterate over each word provided in the input array

// Convert the word to lowercase for uniform comparison

// Determine the keyboard row of the first character of the word

// Initialize an array to store the valid words

const lowerCaseWord = word.toLowerCase();

// Assume the word is valid initially

for (const char of lowerCaseWord) {

isWordValid = false;

// Function to find and return words that can be typed using letters of one row on a keyboard

// The keyboard rows mapped to numbers '1' for the first row, '2' for the second, etc.

const baseRow = keyboardRowMapping[lowerCaseWord.charCodeAt(0) - 'a'.charCodeAt(0)];

// Check each character in the word to see if they are from the same keyboard row

if (keyboardRowMapping[char.charCodeAt(0) - 'a'.charCodeAt(0)] !== baseRow) {

// If the word is valid (all characters are from the same row), add it to the validWords array

// If the character is from a different row, mark the word as invalid and break out of the loop

return filteredWords;

**}**;

**TypeScript** 

for (String word : words) {

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

# Iterate over each word in the provided list.

## implementation step by step:

**Solution Approach** 

Prepare the Keyboard Mapping: The string s "12210111011122000010020202" is crafted such that the position of each character corresponds to a letter in the alphabet (where a is at index 0), and its value ('1', '2', or '3') indicates which row that

letter is on the keyboard. It's a clever encoding because it enables quick access without the need for a more complex data

The solution uses a fairly straightforward algorithm that involves string manipulation and iteration. Here's a walkthrough of the

- structure like a dictionary. Iterate Over Each Word: The main loop of the function goes through each word in the given words list. Normalize the Case: Since the keyboard rows are case-insensitive, each letter of the word is converted to lowercase using .lower().
- Find Row of the First Letter: For each word, the row of the first letter is determined using ord(w[0].lower()) ord('a') to

which of these words can be typed using letters from only one row on an American keyboard.

- find the index in our mapping string s. ord is a built-in function that returns an integer representing the Unicode code point of the character. Thus, ord('a') would be the base Unicode code point for lowercase letters.
- word belong to the same row. The expression s[ord(c.lower()) ord('a')] looks up the row for each character c. If all characters c in w return the same row value as the first letter (x), then all() returns True, and the word is consistent with just one keyboard row. **Store the Valid Word**: If the condition from step 5 is met, the word is appended to the answer list ans.

Return Results: After all words have been checked, the final list ans contains only the words that can be typed using letters

Check Consistency of Rows: We use a generator expression within the all() function to check if all characters in the current

This approach cleverly avoids nested loops and does not require extra space for a more complicated data structure, making the code concise and efficient. **Example Walkthrough** 

Let's consider a small example where our input array of words is ["Hello", "Alaska", "Dad", "Peace"]. We need to determine

**Prepare the Keyboard Mapping**: We have the string s = "12210111011122000010020202", which represents the row numbers of each letter in the alphabet.

Normalize the Case: We convert each word to lowercase. The words are now ["hello", "alaska", "dad", "peace"]. This

Iterate Over Each Word: We begin by examining each word in the array: "Hello", "Alaska", "Dad", and "Peace".

Find Row of the First Letter: Starting with "hello", we find that the first letter h is in the second row using the mapping (i.e., s[ord('h') - ord('a')] gives us '2').

Check Consistency of Rows: We then check each subsequent letter of "hello". We find that e, l, and o are not all in the

Moving to "Alaska": Following the same processing, we determine that the first letter a is in the second row and all subsequent letters of "Alaska" (1, a, s, k, a) are also in the second row. This means that "Alaska" qualifies.

Processing "Dad": The first letter d is in the second row, but the second letter a is also in the second row, and the next d is in

- **Last Word "Peace"**: We see that the first letter p is in the first row. However, the next letters e, a, and c are not on the first row. Therefore, "Peace" does not qualify.
- Solution Implementation

Return Results: We return the list ["Alaska", "Dad"], as these are the words from the original input that can be typed using

class Solution:

# Get the row for the first character of the word (convert to lowercase for uniformity).

if all(row\_mapping[ord(char.lower()) - ord('a')] == first\_char\_row for char in word):

# Return the list of valid words that can be typed using letters of one row on the keyboard.

# Mapping for each letter to its corresponding row on the keyboard.

# If they are, append the word to our list of valid words.

// Function to find the words that can be typed using letters of one row on the keyboard

// String representing rows of the keyboard as numbers (1st row, 2nd row, 3rd row)

// a, b, c, etc. are mapped to their respective row numbers

// Convert the word to lower case to make it case—insensitive

char initialRow = rowMapping.charAt(lowerCaseWord.charAt(0) - 'a');

String rowMapping = "12210111011122000010020202";

String lowerCaseWord = word.toLowerCase();

// Iterate over characters of the word

// Get the row number of the first character

// Flag to check if all letters are in the same row

for (char character : lowerCaseWord.toCharArray()) {

Store the Valid Words: We have found that the words "Alaska" and "Dad" both meet the criteria.

def findWords(self, words: List[str]) -> List[str]: # Initialize an empty list to store the valid words. valid\_words = []

### first\_char\_row = row\_mapping[ord(word[0].lower()) - ord('a')] # Check if all characters in the word are in the same row as the first one.

for word in words:

return valid\_words

```
Java
class Solution {
```

10.

**Python** 

```
// Check if character is in a different row
                if (rowMapping.charAt(character - 'a') != initialRow) {
                    canBeTyped = false;
                    break; // No need to check further if one letter is in a different row
           // If all characters are in the same row, add the original word to the result
            if (canBeTyped) {
                result.add(word);
        // Return the result as an array of strings
        return result.toArray(new String[0]);
C++
#include <vector>
#include <string>
#include <cctype> // Necessary for tolower function
class Solution {
public:
    // Function to find the words that can be typed using letters of one row on the keyboard
    vector<string> findWords(vector<string>& words) {
       // Map each alphabet to a number corresponding to the row in the keyboard.
       // 1 for first row, 2 for second row and so on.
        string rowIndices = "12210111011122000010020202";
        vector<string> filteredWords; // Resultant vector of words
        // Iterate over each word in the input list
        for (auto& word : words) {
            bool canBeTyped = true; // flag to check if the word belongs to a single row
           // Get the row for the first character of the word
            char initialRow = rowIndices[tolower(word[0]) - 'a'];
            // Iterate over each character in the word
            for (char& character : word) {
                // If the character does not belong to the same row as the first character, set flag to false
                if (rowIndices[tolower(character) - 'a'] != initialRow) {
                    canBeTyped = false;
                    break; // Break out of the character loop since one mismatch is enough
           // If the word can be typed using letters of one row, add it to the result
            if (canBeTyped) {
                filteredWords.emplace_back(word);
```

```
validWords.push(word);
```

```
// Return the list of valid words
      return validWords;
class Solution:
   def findWords(self, words: List[str]) -> List[str]:
       # Initialize an empty list to store the valid words.
       valid words = []
       # Mapping for each letter to its corresponding row on the keyboard.
        row_mapping = "12210111011122000010020202"
       # Iterate over each word in the provided list.
       for word in words:
           # Get the row for the first character of the word (convert to lowercase for uniformity).
           first_char_row = row_mapping[ord(word[0].lower()) - ord('a')]
           # Check if all characters in the word are in the same row as the first one.
           if all(row_mapping[ord(char.lower()) - ord('a')] == first_char_row for char in word):
               # If they are, append the word to our list of valid words.
               valid words.append(word)
       # Return the list of valid words that can be typed using letters of one row on the keyboard.
       return valid_words
Time and Space Complexity
  The given Python function findWords filters a list of words, selecting only those that can be typed using letters from the same row
  on a QWERTY keyboard.
Time Complexity:
```

### The time complexity of the function can be considered as O(N \* M) where: • N is the number of words in the input list words. • M is the average length of the words.

- keyboard rows to characters. The comparison s[ord(c.lower()) ord('a')] == x runs in 0(1) time since it's a simple constanttime index access and comparison.
- Therefore, we need to perform M constant-time operations for each of the N words, which leads to an overall time complexity of

0(N \* M).**Space Complexity:** 

For each word, the function checks if all characters belong to the same row by referencing a pre-computed string s that maps

• N is the number of words in the input list words. • K is the average space taken by each word.

The space complexity is O(N \* K) where:

The space complexity owes to the storage of the output list ans. Each word that meets the condition is appended to ans. In the worst case, all N words meet the condition, and we end up storing all of them in ans, hence the O(N \* K) space complexity.

The string s used for row mapping is constant and does not scale with the input, so it does not add to the space complexity.