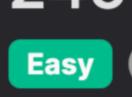
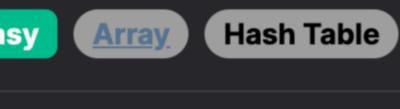
integer array, our job is to find which one doesn't fit with the others.





Problem Description

In this problem, you are given an array of strings words, where all strings are of equal length n. We can imagine that each string is like a sequence of positions from a starting point in the alphabet. For example, "acd" could mean starting at 'a' (position 0), moving to 'c' (position 2), and then to 'd' (position 3).

For each string, we can calculate a difference integer array. This array represents the difference between each adjacent pairs of characters in the alphabet. So, for a string words [i], we calculate difference[i][j] as the difference in the alphabet positions

between words[i][j+1] and words[i][j], for all j from 0 up to n - 2. Example: For the string "acd", the difference integer array would be [2, 1] because that's the difference between 'c' and 'a' (2) and 'd' and 'c' (1). The key point to remember is that we're only looking at the relative differences between neighboring characters, not

difference integer array does not match with the others. Intuition

All strings in the array words have the same difference integer array except for one string. Your task is to return the string whose

The solution works on the principle of grouping and counting. Since we know that all strings except one will form the same difference

absolute positions.

How can we find the odd one out? We can use a defaultdict from Python's collections module to group the strings based on their difference integer arrays. We calculate the difference integer array for each string, use it as a key, and append the string to the

associate list in the dictionary. Since the difference integer array is used for grouping, we need it to be a hashable type, hence we create a tuple (t) from it. For example, if "abc" and "def" both yield a difference integer array of [1, 1], they will be added to the same list in the dictionary under the key (1, 1).

After we group all the strings, we look through the dictionary's values. The odd string out will be the only string in its group, so we look for any list in the dictionary with a length of 1. This list contains our odd string, and that's the one we return.

In summary, by calculating the difference integer arrays for each string, grouping them by these arrays, we can quickly identify which string has a unique difference pattern, as it will be alone in its group.

The provided solution follows these steps to find the string that has a different difference integer array:

1. Importing Necessary Modules: The solution begins by importing defaultdict from the collections module, which allows for

easy grouping without having to initialize empty lists manually for each new key.

Solution Approach

2. Compute the Difference Tuples: It iterates over each string s in the words array and computes the difference tuple t using a

generator expression. The expression ord(b) - ord(a) is used for each adjacent pair (a, b) of characters in the string, where ord() is a built-in Python function that returns the ASCII value (or position in the alphabet for lowercase letters) of the character. 3. Group Strings by their Difference Tuples: Using the difference tuple t as a key, the solution then appends the current string s to

a list in the dictionary d. This effectively groups all strings that produce the same difference tuple together.

operations, where n is the length of the strings. The lookup and insertion times for a defaultdict are typically O(1).

Defaultdict: A convenient way to group strings without manual checks for the existence of dictionary keys.

• Next Function: A pythonic way to retrieve the first element of an iterator that satisfies a certain condition.

- 4. Find the Unique String: Finally, the solution uses a generator expression with a next() function to iterate over the dictionary's values. It searches for the first list ss that has a length of 1, indicating that it contains the unique string. Once it finds such a list, it retrieves the first (and only) string from that list, which is the string with a different difference integer array.
- Notably, the code uses pairwise from the itertools module, which is not directly available in the provided code. If this function were available or implemented, it would pair adjacent elements of s, facilitating the calculation of differences.

The algorithm complexity is primarily determined by the iteration over the strings and the difference calculation, which are both O(n)

Here's a breakdown of the data structures and patterns used: • Generator Expressions: Used to efficiently compute difference tuples without needing intermediate lists.

The solution effectively uses these Python features to group strings and find the unique one in a concise and expressive manner.

Tuples: Immutable and hashable, allowing them to be used as dictionary keys for grouping.

- Example Walkthrough
- 1. First, we compute the difference integer arrays (which will be stored as tuples) for each string. These tuples represent the differences in position between each pair of adjacent characters:

Let's take a set of strings words = ["abc", "bcd", "ace", "xyz", "bbb"] as a small example to illustrate the solution approach.

For "ace": differences are (c-a, e-c), which is (2, 2).

(2, 2): ["ace"],

(0, 0): ["bbb"]

Each string has a length of n = 3.

 For "xyz": differences are (y-x, z-y), which is (1, 1). For "bbb": differences are (b-b, b-b), which is (0, 0).

2. We then group the strings by difference tuples using a defaultdict, resulting in a dictionary with difference tuples as keys and

```
3. By inspecting the dictionary, we can see the group that contains only a single string is the one with the key (2, 2), which
  corresponds to the string "ace".
4. Therefore, by using a generator expression with the next() function to search for the first occurrence of such a one-string
  group, we find that "ace" is the string with a different difference integer array.
```

For "abc": differences are (b-a, c-b), which is (1, 1).

For "bcd": differences are (c-b, d-c), which is (1, 1).

lists of corresponding strings as values:

5. "ace" is returned as the odd one out string.

from collections import defaultdict

for word in words:

pattern_to_words = defaultdict(list)

Iterate over each string in the list of words

of the ASCII values of characters

pattern_to_words[pattern].append(word)

from itertools import pairwise

(1, 1): ["abc", "bcd", "xyz"],

Python Solution

string whose difference integer array does not match with the others in a clear and concise way.

Compute the pattern of the word based on the pairwise differences

Append the original word to the list of words for this pattern

If no word is found that satisfies the condition, i.e., being the only one

// Convert the difference array to a string to use as a key in the map.

// Add the current word to the list corresponding to its difference sequence.

// If a particular difference sequence is unique to one word, return that word.

// If the key is not present in the map, create a new list for it.

differenceMap.putIfAbsent(differenceKey, new ArrayList<>());

for (List<String> wordsWithSameDifference : differenceMap.values()) {

String differenceKey = String.valueOf(differenceArray);

differenceMap.get(differenceKey).add(word);

if (wordsWithSameDifference.size() == 1) {

return wordsWithSameDifference.get(0);

// If no "odd" string is found, return an empty string.

// Iterate over the entries in the map.

of its pattern, (which theoretically shouldn't happen given the problem's constraints),

an explicit return is not required as functions return None by default when no return is hit

So, using this grouping and counting strategy with efficient Python data structures and expressions, we identified that "ace" is the

class Solution: def oddString(self, words: List[str]) -> str: # Create a dictionary to map string patterns to strings

pattern = tuple(ord(second_char) - ord(first_char) for first_char, second_char in pairwise(word))

```
# Iterate over each list of words associated with the same pattern
           for words_with_same_pattern in pattern_to_words.values():
               # If a list contains only one word, return this word as it is the odd one out
               if len(words_with_same_pattern) == 1:
22
                   return words_with_same_pattern[0]
```

13

14

15

16

17

18

23

24

25

26

27

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

42

43

44

45

46

47

```
Java Solution
   class Solution {
       /**
        * Method to find the "odd" string in a given array of words. The "odd" string is defined as the
        * one which doesn't have any other string in the array with the same sequence of differences
        * between consecutive characters.
        * @param words An array of strings to process.
        * @return The "odd" string if it exists, or an empty string otherwise.
9
10
       public String oddString(String[] words) {
11
           // Create a dictionary to map the difference sequence to a list of strings that share it.
12
13
           HashMap<String, List<String>> differenceMap = new HashMap<>();
14
15
           // Iterate over each word in the array.
           for (String word : words) {
16
17
               int length = word.length();
18
               // Create an array to store the differences in ASCII values between consecutive characters.
               char[] differenceArray = new char[length - 1];
20
21
22
               for (int i = 0; i < length - 1; ++i) {
23
                   // Calculate the difference and store it in the array.
24
                   differenceArray[i] = (char) (word.charAt(i + 1) - word.charAt(i));
25
```

48 }

return "";

```
49
C++ Solution
1 #include <vector>
2 #include <string>
   #include <unordered_map>
  class Solution {
  public:
       // This function finds and returns the word from the given words vector that has no matching pattern
       string oddString(vector<string>& words) {
           // Dictionary to map a pattern to a list of words that share the same pattern
           unordered_map<string, vector<string>> patternToWords;
10
12
           // Process each word in the vector
13
           for (auto& word : words) {
               int length = word.size(); // Length of the current word
14
15
               string pattern(length - 1, 0); // Create a pattern string
16
               // Create a pattern based on the difference in ASCII values
17
               // between consecutive characters in the word
19
               for (int i = 0; i < length - 1; ++i) {
20
                   // Subtract consecutive characters and store it in the pattern
                   pattern[i] = word[i + 1] - word[i];
21
22
               // Add the word to the dictionary based on its pattern
24
25
               patternToWords[pattern].push_back(word);
26
27
28
           // Iterate through the mapped dictionary to find a unique pattern
29
           for (auto& kv : patternToWords) { // kv is a pair consisting of a pattern and a list of words
               auto& wordsWithSamePattern = kv.second; // Get the list of words with the same pattern
30
31
32
               // If there's only one word with this pattern, that's our "odd" word
33
               if (wordsWithSamePattern.size() == 1) {
                   return wordsWithSamePattern[0];
34
35
36
37
38
           // If no unique pattern is found, return an empty string
39
           return "";
40
41 };
42
Typescript Solution
```

24 patternMap.get(pattern)!.push(word); 25 26 27 // Find the odd string out i.e., a string whose pattern is unique for (const wordGroup of patternMap.values()) { 28

return '';

10

11

12

13

14

15

16

17

18

19

20

21

23

30

31

32

33

34

35

37

36 }

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

// Iterate over each word

for (const word of words) {

const patternMap: Map<string, string[]> = new Map();

for (let i = 0; i < word.length - 1; ++i) {

const pattern = charDifferences.join(',');

// If there's no odd string found, return an empty string

if (!patternMap.has(pattern))

return wordGroup[0];

Time and Space Complexity

patternMap.set(pattern, []);

const charDifferences: number[] = [];

// Array to hold the character difference pattern

// Map to keep track of strings with the same character difference pattern

// Compute consecutive character differences for the current word

charDifferences.push(word.charCodeAt(i + 1) - word.charCodeAt(i));

// If the pattern is not already in the map, initialize it with an empty array

if (wordGroup.length === 1) { // The odd string will be the only one in its group

// Add the current word to the array of words that match the same pattern

// Convert the character differences to a string, joining with commas

Time Complexity The given code block consists of the following operations:

words in words.

Space Complexity

each iteration.

• Inside the loop, for each word, it calculates the tuple t using the pairwise function and a list comprehension, which iterates through each consecutive pair of characters in the word. If the average length of the words is M, this part has a complexity of

be up to N unique tuples if all words are distinct.

O(M). Therefore, the loop overall has a complexity of O(N * M). A next function is then used with a generator expression to find the first tuple t with only one associated word in the dictionary

d. In the worst case scenario, this operation will go through all the tuples generated, with a complexity of O(N), since there could

A loop that iterates through each word in the input list words: This will have an O(N) complexity, where N is the total number of

- As a result, the total time complexity of this code block would be O(N * M) + O(N), which simplifies to O(N * M) since M (the average length of a single word) is usually much smaller than N or at least M is considered constant for relatively short strings compared to the
- number of words.

• The dictionary d which stores tuples of integers as keys and lists of strings as values. In the worst-case scenario, if all words

The space complexity is affected by:

- have unique difference tuples, the space used would be O(N * M), where M is the average length of word lists associated with each tuple. However, the tuples and words themselves just keep references to already existing strings and characters, so we can consider that storing the tuples would at worst be O(N). • The space used by the list comprehension inside the loop does not add to the space complexity since it's used and discarded at
- Thus, the space complexity would be dominated by the dictionary d storing elements. So the space complexity of the code is O(N).