

**Problem Description** 

The problem presents us with an array of strings called wordsDict, where we must find the minimum distance between two specified words, word1 and word2. This distance is defined as the number of positions between the two words in the array. An important condition given is that word1 and word2 can sometimes be the same word in the list, although generally, they represent distinct words. The goal is to write a function that returns the smallest number of indices between the two occurrences of these

## words.

Intuition

The intuitive approach to solving this problem relies on a linear scan through the wordsDict array. We will keep track of the positions where word1 and word2 occur as we iterate through the array. The key is to update the minimum distance whenever we encounter one of the two words.

If word1 and word2 are the same, we need to find the smallest distance between two occurrences of this single word. We can do this by keeping a single pointer (let's call it j) that records the last seen position of the word. As we iterate, each time we encounter the word again, we calculate the distance from the current position to the last seen position and update our answer accordingly.

we've previously seen the other word. If we have, we calculate the distance between the current word's position and the last seen position of the other word and update the minimum distance accordingly. **Solution Approach** 

positions for both words. As we move through the array, if we find either word, we update the respective pointer and check if

On the other hand, if word1 and word2 are different, we maintain two separate pointers (say i and j) to track the last seen

## the words and calculate the minimum distance. Here's a step-by-step breakdown of how the algorithm operates:

than the current ans.

to the above logic.

where n is the number of words in wordsDict.

Initialization: A variable ans is initialized with the length of wordsDict. This is done because the maximum possible distance between any two words is the length of the array itself. We also initialize two pointers i and j to -1 to represent that we have

The solution implemented here makes use of a few key variables and a single pass through the input list to track the positions of

Special Case Handling for Identical Words: When word1 and word2 are the same, they will be encountered sequentially at

not yet encountered any of the two words.

- different positions as we iterate. We only need one index j to keep track of the last position where this word was found. Each time we find the word again, we calculate the distance between the current and last positions and update ans if it is smaller
- General Case for Different Words: Here, we need to track the most recent positions of word1 and word2 as we iterate. Whenever one of the words is found (w == word1 or w == word2), we update the respective last seen index, i or j. If both i and j have been set (meaning we have found both words at least once), we calculate the absolute difference between their positions to get the current distance. We then update ans if the calculated distance is less than the current ans.

Iteration: The loop goes through every word win wordsDict with its index k and performs the checks and updates according

**Return Statement:** After the loop completes, ans contains the smallest distance found between the two words, and this value is returned. This solution is elegant because it covers both cases (identical and different words) in a single pass and does so with constant

space complexity, i.e., the space used does not grow with the input size, and linear time complexity, i.e., it runs in O(n) time,

**Example Walkthrough** 

"perfect", "coding", "makes"], and we want to find the minimum distance between word1 = "coding" and word2 = "practice". Initialization: ans is set to the length of wordsDict, which is 5.

○ Two pointers i and j are initialized to -1 as we have not yet found any instance of word1 or word2.

The final word in the array does not match either word1 or word2, so no further actions are taken.

• We go through each word w in wordsDict along with its index k.

Let us consider a small example to illustrate the solution approach. Assume the array wordsDict is ["practice", "makes",

**Return Statement:** 

**Iteration:** 

○ At index 1, w = "makes". This does not match word1 or word2, so we continue onwards without updates. • At index 3, w = "coding". This is identified as word2, so we update j to 3. Now, we check for i, which is 0. We find the distance between index 3 and index 0 to be 3 and update ans with this value because it's smaller than the current ans.

Since we have now finished iterating through wordsDict, the ans variable contains the smallest distance, which is 3. Thus, the function will

If we alter the conditions slightly and look for the minimum distance between the same word, word1 = "makes" and word2 =

At index 0, w = "practice". We recognize this as word1 and update i to 0. Since j is still −1, we do not update ans.

"makes", we would only need one pointer as per the special case in our approach.

Iteration:

Initialization:

○ ans set to 5 and j set to -1.

**Return Statement:** 

is 3, and update our answer to 3.

return 3.

- At index 1, w = "makes". We update j to 1 (there's nothing to compare with yet, so ans remains unchanged). o Continuing through the array, we find "makes" again at index 4. Now, we check the distance from the last position 1 to the current 4, which
- After completing the loop, the final ans is 3, as that is the minimum distance between two instances of "makes". Solution Implementation
- def shortest\_word\_distance(self, words\_dict: List[str], word1: str, word2: str) -> int: # Initialize the minimum distance to the length of the word list min\_distance = len(words\_dict)

class Solution:

from typing import List

**Python** 

 $last_occurrence_index = -1$  # Initialize the index for the last occurrence of the word for index, word in enumerate(words\_dict): if word == word1:

# Update minimum distance if the current pair is closer

# Update the last occurrence index to the current index

index1 = index # Update index when word1 is found

index2 = index # Update index when word2 is found

// Function to find the shortest distance between two words in a dictionary,

int dictSize = wordsDict.size(); // Get the size of the dictionary.

for (int i = 0, lastPosition = -1; i < dictSize; ++i) {

// Loop through each word in the dictionary.

// from the last occurrence.

// Update last occurrence position.

// Loop through each word in the dictionary.

if (positionWord1 != -1 && positionWord2 != -1) {

// Function to find the shortest distance between two words in a dictionary,

let lastPosition: number = -1; // Initialize the last position.

const dictSize: number = wordsDict.length; // Get the size of the dictionary.

answer = min(answer, i - lastPosition);

int shortestWordDistance(vector<string>& wordsDict, string word1, string word2) {

int answer = dictSize; // Initialize answer with the maximum possible distance.

// When the word matches word1 (which is also word2 in this case).

for (int k = 0, positionWord1 = -1, positionWord2 = -1; k < dictSize; ++k) {

// When the word matches word1, update its last seen position.

// When the word matches word2, update its last seen position.

// If both words have been seen at least once, calculate the distance.

answer = min(answer, abs(positionWord1 - positionWord2));

function shortestWordDistance(wordsDict: WordDictionary, word1: string, word2: string): number {

let answer: number = dictSize; // Initialize answer with the maximum possible distance.

// If this is not the first occurrence, calculate the distance

// where the words could potentially be the same.

// Case when both words are the same.

if (wordsDict[i] == word1) {

lastPosition = i;

// Case when the words are different.

if (wordsDict[k] == word1) {

} else if (wordsDict[k] == word2) {

positionWord1 = k;

positionWord2 = k;

// Return the shortest distance found.

// Import the necessary utilities from Array.prototype

// Type definition for a dictionary of words.

// where the words could potentially be the same.

// Case when both words are the same.

wordsDict.forEach((word, index) => {

if (lastPosition != -1) {

**if** (word1 == word2) {

# If both words have been found, update the minimum distance

min\_distance = min(min\_distance, index - last\_occurrence\_index)

# If both words are the same, we need to find the shortest distance

# between two occurrences of the same word

if last\_occurrence\_index != -1:

last\_occurrence\_index = index

for index, word in enumerate(words\_dict):

if word == word1:

if word == word2:

```
else:
    # If the words are different, find the shortest distance between word1 and word2
    index1 = index2 = -1 # Initialize the indexes for the occurrences of the words
```

if word1 == word2:

```
if index1 != -1 and index2 != -1:
                   min_distance = min(min_distance, abs(index1 - index2))
       # Return the minimum distance found
       return min_distance
Java
class Solution {
   // Function to find the shortest distance between occurrences of word1 and word2 in wordsDict
    public int shortestWordDistance(String[] wordsDict, String word1, String word2) {
        int shortDistance = wordsDict.length; // Initialize to the maximum possible distance
       // Handle the case where both words are the same
       if (word1.equals(word2)) {
            for (int i = 0, prevIndex = -1; i < wordsDict.length; ++i) {
                if (wordsDict[i].equals(word1)) {
                    if (prevIndex != -1) {
                        // Update the shortest distance between two occurrences of word1
                        shortDistance = Math.min(shortDistance, i - prevIndex);
                   prevIndex = i; // Update prevIndex to the current index
       } else {
           // When the two words are different
            for (int k = 0, indexWord1 = -1, indexWord2 = -1; k < wordsDict.length; ++k) {
                if (wordsDict[k].equals(word1)) {
                    indexWord1 = k; // Update index for word1
               if (wordsDict[k].equals(word2)) {
                    indexWord2 = k; // Update index for word2
               // If both indices are set, calculate distance and update if it's a shorter one
               if (indexWord1 !=-1 && indexWord2 !=-1) {
                    shortDistance = Math.min(shortDistance, Math.abs(indexWord1 - indexWord2));
       return shortDistance; // Return the shortest distance found
```

**}**;

**TypeScript** 

} else {

return answer;

import { abs, min } from "Math";

type WordDictionary = string[];

**if** (word1 === word2) {

C++

#include <vector>

#include <string>

#include <climits>

using std::vector;

using std::string;

using std::min;

using std::abs;

class Solution {

public:

```
// Loop through each word in the dictionary.
if (word === word1) {
    // When the word matches word1 (which is also word2 in this case).
```

```
if (lastPosition !== -1) {
                      // If this is not the first occurrence, calculate the distance
                      // from the last occurrence.
                      answer = min(answer, index - lastPosition);
                  // Update last occurrence position.
                  lastPosition = index;
          });
      } else {
          // Case when the words are different.
          let positionWord1: number = -1; // Initialize position for word1.
          let positionWord2: number = -1; // Initialize position for word2.
          wordsDict.forEach((word, index) => {
              // Loop through each word in the dictionary.
              if (word === word1) -
                  // When the word matches wordl, update its last seen position.
                  positionWord1 = index;
              } else if (word === word2) {
                  // When the word matches word2, update its last seen position.
                  positionWord2 = index;
              // If both words have been seen at least once, calculate the distance.
              if (positionWord1 !== -1 && positionWord2 !== -1) {
                  answer = min(answer, abs(positionWord1 - positionWord2));
          });
      // Return the shortest distance found.
      return answer;
from typing import List
class Solution:
   def shortest_word_distance(self, words_dict: List[str], word1: str, word2: str) -> int:
       # Initialize the minimum distance to the length of the word list
       min_distance = len(words_dict)
       # If both words are the same, we need to find the shortest distance
       # between two occurrences of the same word
       if word1 == word2:
            last\_occurrence\_index = -1 # Initialize the index for the last occurrence of the word
            for index, word in enumerate(words_dict):
               if word == word1:
                    if last_occurrence_index != -1:
                        # Update minimum distance if the current pair is closer
                        min_distance = min(min_distance, index - last_occurrence_index)
                   # Update the last occurrence index to the current index
                    last_occurrence_index = index
       else:
           # If the words are different, find the shortest distance between word1 and word2
            index1 = index2 = -1 # Initialize the indexes for the occurrences of the words
            for index, word in enumerate(words_dict):
               if word == word1:
                    index1 = index # Update index when word1 is found
               if word == word2:
                    index2 = index # Update index when word2 is found
               # If both words have been found, update the minimum distance
               if index1 != -1 and index2 != -1:
                   min_distance = min(min_distance, abs(index1 - index2))
       # Return the minimum distance found
       return min_distance
Time and Space Complexity
```

## **Time Complexity** The given algorithm traverses through the list wordsDict once, regardless of whether word1 and word2 are the same or not.

When word1 and word2 are the same, the algorithm iterates over wordsDict, and only performs constant-time operations

- within the loop (comparison, assignment, arithmetic). Thus, the time complexity is O(n), where n is the length of wordsDict. When word1 and word2 are different, the algorithm again only goes through the list once, doing constant-time operations •
  - whenever word1 or word2 is found, and updating indices i and j. The time complexity remains 0(n).

The space complexity is 0(1). The algorithm uses a fixed number of integer variables (ans, i, j, k) and these do not depend on

**Space Complexity** 

the size of the input (wordsDict). There are no data structures that grow with the size of the input.