244. Shortest Word Distance II

Feb. 10, 2019 | 23.3K views
Average Rating: 4.21 (26 votes)

Design a class which receives a list of words in the constructor, and implements a method that takes two words *word1* and *word2* and return the shortest distance between these two words in the list. Your method will be called *repeatedly* many times with different parameters.

Example:
Assume that words = `["practice", "makes", "perfect", "coding", "makes"]`.

```
Input: word1 = "coding", word2 = "practice"
Output: 3
```

```
Input: word1 = "makes", word2 = "coding"
Output: 1
```

Note:
You may assume that word1 **does not equal to** word2, and word1 and word2 are both in the list.

Solution

Before looking at the solution for this problem, let's look at what the problem asks us to do in simpler terms. We have to design a class which receives a list of words as input in the constructor. The class has a function which we need to implement and that function is `shortest` which takes two words as input and returns the minimum distance between the two as the output.

When the problem talks about the distance between two words, it essentially means the absolute gap between the indices of the two words in the list. For eg. if the first word occurs at a location `i` and the second word occurs at the location `j`, then the distance between the two would be `abs(i - j)`.

The question asks us to find the `minimum` such different between words which clearly indicates that the words can occur at multiple locations. If we have K occurrences for the `word1` and `L` occurrences for the `word2`, then iteratively checking every pair of indices will give us a $O(N^2)$ algorithm which won't be optimal at all. We won't discuss that algorithm here since it is very straightforward.

The brute-force algorithm would simple consider all possible pairs of indices for (`word1_location`, `word2_location`) and see which one produces the minimum distance. Let's try and build on this idea and see if some pre-processing can help us cut reduce the complexity of the brute-force algorithm.

Approach 1: Using Preprocessed Sorted Indices

Intuition

A given word can occur multiple times in the original word list. Let's suppose the first word, `word1`, in the input to the function `shortest` occurs at the indices `[i1, i2, i3, i4]` in the original list. Similarly, let's assume that the second word, `word2`, appears at the following locations inside the word list `[j1, j2, j3]`.

Now, given these list of indices, we are to simply find the pair of indices `(i, j)` such that their absolute difference is minimum.

> The main idea for this approach is that if the list of these indices is in sorted order, we can find such a pair in linear time.

The idea is to use a two pointer approach. Let's say we have a pointer `i` for the sorted list of indices of `word1` and `j` for the sorted list of indices of `word2`. At every iteration, we record the difference of indices i.e. `abs(word1[i] - word2[j])`. Once we've done that, we have two possible choices for progressing the two pointers.

```
word1[i] < word2[j]
```

If this is the case, that means there is no point in moving the `j` pointer forward. The location indices for the words are in a sorted order. We know that `word1[i + 1] > word2[j]` because these indices are sorted. So, if we move `j` forward, then the difference `abs(word1[i] - word2[j + 1])` would be even greater than `abs(word1[i] - word2[j])`. That doesn't help us since we need the minimum possible distance (difference) overall.

> So, if we have (word1[i] < word2[j]), we move the pointer 'i' one step forward i.e. (i = 1) in the hopes that abs(word1[i + 1] - word2[j]) would give us a lower distance than abs(word1[i] - word2[j]). We say "hopes" because it is not certain this improvement would happen.

Let's look at two different examples. In the first example we will see that moving `i` forward gave us the best difference overall (0). In the second example we see that moving `i` forward leads us to our second case (yet to discuss) but doesn't lead to any improvement in the difference.

Example-1

```
word1_locations = [2,4,5,9]
word2_locations = [4,10,11]

i, j = 0, 0
min_diff = 2 (abs(2 - 4))
word1[i] < word2[j] i.e. 2 < 4
  move i one step forward

i, j = 1, 0 (abs(4 - 4))
min_diff = 0 (We hit the jackpot!)
```

Example-2

```
word1_locations = [2,7,15,16]
word2_locations = [4,10,11]

i, j = 0, 0
min_diff = 2 (abs(2 - 4))
word1[i] < word2[j] i.e. 2 < 4
  move i one step forward

i, j = 1, 0
min_diff = 2 (2 < abs(7 - 4))

Here, we did not update out global minimum difference.
That is why we said earlier, moving 'i' forward may or
may not give a lower difference. But moving 'j' forward in
our case would definitely worsen the difference (or keep it same!).
```

Let's move onto our second scenario.

```
word1[i] > word2[j]
```

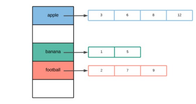
If this is the case, that means there is no point in moving the `j` pointer forward. We know that `word1[i + 1] > word2[j]` because these indices are sorted. So, if we move `j` forward, then the difference `abs(word1[i + 1] - word2[j])` would be even greater than `abs(word1[i] - word2[j])`. That doesn't help us since we want to find the minimum possible distance (difference) overall.

> So, along the similar lines of thought as the previous case, if we have (word1[i] > word2[j]), we move the pointer 'j' one step forward i.e. (j = 1) in the hopes that abs(word1[i - word2[j + 1]) would give us a lower distance than abs(word1[i] - word2[j]). We say "hopes" because as showcased in the previous scenario, it is not certain this improvement would happen.

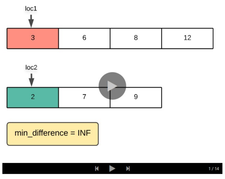
Now let's formally look at the algorithm for solving this problem.

Algorithm

1. In the `constructor` of the class, we simply iterate over the given list of words and prepare a dictionary, mapping a word to all it's locations in the array.
2. Since we process all the words from left to right, we will get all the indices in a sorted order by default for all the words. So, we don't have to sort the indices ourselves.
3. Let's call the dictionary that we build, `locations`.
4. For a given pair of words, obtain the list of indices (appearances inside the original list/array of words). Let's call the two arrays `loc1` and `loc2`.
5. Initialize two pointer variables `i1 = 0` and `i2 = 0`.
6. For a given `i1` and `i2`, we first update (if possible) the minimum difference (distance) till now i.e. `dist = min(dist, abs(loc1[i1] - loc2[i2]))`. Then, we check if `loc1[i1] < loc2[i2]` and if this is the case, we move `i1` one step forward i.e. `i1 = i1 + 1`. Otherwise, we move `i2` one step forward i.e. `i2 = i2 + 1`.
7. We keep doing this until all the elements in the smaller of the two location arrays are processed.
8. Return the global minimum distance between the words.



This represents the locations dictionary that we should build given the original words list in the constructor. The key represents the word and the value is a list containing indices in ascending order of occurrences throughout the array. Let's look at the minimum distance between the words `apple` and `football` in the array. So, we will be considering the two *sorted* lists of indices: `[3, 6, 8, 12]` and `[2, 7, 9]`.




```
Java   Python                                                    Copy
from collections import defaultdict
class WordDistance:

    def __init__(self, words):
        """
        :type words: List[str]
        """
        self.locations = defaultdict(list)

        # Preparing a mapping from a word to all it's locations (indices).
        for w, i in enumerate(words):
            self.locations[w].append(i)

    def shortest(self, word1, word2):
        """
        :type word1: str
        :type word2: str
        :rtype: int
        """
        loc1, loc2 = self.locations[word1], self.locations[word2]
        i1, i2 = 0, 0
        min_diff = float('inf')

        # Until the shorter of the two lists is processed
        while i1 < len(loc1) and i2 < len(loc2):
            min_diff = min(min_diff, abs(loc1[i1] - loc2[i2]))
            if loc1[i1] < loc2[i2]:
```

Complexity analysis

- Time complexity : The time complexity of the constructor of our class is $O(N)$ considering there were N words in the original list. We iterate over them and prepare a mapping from key to list of indices as described before. Then, the function finds the minimum distance between the two words, the complexity would be $O(\max(K, L))$ where K and L represent the number of occurrences of the two words. However, $K = O(N)$ and also $L = O(N)$. Therefore, the overall time complexity would also be $O(N)$. The reason the complexity is $O(\max(K, L))$ and not $O(\min(K, L))$ is because of the scenario where the minimum element of the smaller list is larger than `all` the elements of the larger list. In that scenario, the pointer for the smaller list will not progress at all and the one for the longer list will reach to the very end.
- Space complexity : $O(N)$ for the dictionary that we prepare in the constructor. The keys represent all the unique words in the input and the values represent all of the indices from $0 \ldots N$.

Rate this article:

Previous Next

Comments: 10 Sort By

Type comment here... (Markdown is supported)
Preview Post

yijiang2023 ★ 15 March 23, 2019 8:20 PM
thanks for your article.
I believe the function that finds the minimum distance between the 2 words should be O(k + l).
consider the following case, K = L:
word1 = [1 3 5 7 9]
word2 = [2 4 6 8 10]
Read More
14 Share Reply
SHOW 1 REPLY

ka19 ★ 402 April 30, 2019 9:07 PM
Just a slight mistake in the article, in the first example, word1 and word 2 seem to happen at the same index 4 which can never happen according to the problem statement
12 Share Reply
SHOW 1 REPLY

code_monkey ★ 10 December 10, 2019 5:09 AM
poorly stated problem. The problem definition is highly incomplete.
5 Share Reply
SHOW 1 REPLY

pyfyfan ★ 131 July 29, 2019 9:56 AM
This method makes sense, but why can't we just repeat the same one in Shortest Word Distance I. We just store the whole words list in the constructor, and do the linear time method to get the min distance in 'shortest' function. The overall space complexity and time complexity are both O(N), which is same as the answer above.
Read More
4 Share Reply
SHOW 2 REPLIES

vrushetnikov ★ 15 March 25, 2019 10:49 AM
To find the leftmost occurrence of the other word to the right of the current position of the current word we can use a binary search (`upper_bound`) instead of a linear search. Here is a C++ implementation that runs in 35ms (faster than 100% of other C++ submissions):
Read More
2 Share Reply
SHOW 1 REPLY

zhang10 ★ 3 March 18, 2020 12:31 PM
I used binary search with iteration for querying minimal distance, whose time complexities is O(min(K,L)log(max(K,L)). I think it will be better in cases when L >> K.
1 Share Reply
SHOW 1 REPLY

Yingtao ★ 1 June 4, 2019 9:07 AM
I think having a memo map to record calculated results could be practically temporally more efficient though this would lead to O(N2) space complexity.
0 Share Reply

hunsdi ★ 22 April 4, 2019 11:53 AM
The code is correct, but the comment is wrong. "Until the shorter of the two lists is processed", not necessarily. The longer list could be consumed first.
0 Share Reply

hegde0hai ★ 8 3 days ago
I tried doing this just by using hashmap<String,Integer>, of course should've thought of repeated words, then changed to HashMap<String,Array<int>>. nice question though
0 Share Reply

layak ★ 3 January 30, 2020 4:58 AM
Can someone explain what is the difference between shortest-word-distance and shortest-word-distance-ii ? It seems the same to me but solution seem different
0 Share Reply
SHOW 1 REPLY