In this question, I did the z-algorithm twice - one for the pattern match of the original string and another one for the reversed string. For the original string, I join the pattern with the text with the '$' symbol as an indication to separate the pattern and the text. Meanwhile, for the reversed string, I reversed the pattern and the text but still joined them with '$'. The complexity for reversing the string is O(n+m) or O(n) assuming that the size of text is always bigger than the pattern.

The combine_text_pattern is used to concatenate the string with '$'. The function has a complexity of O(n+m) and an auxiliary space complexity of O(1) as they only store 2 values.

The string_match function is the simplest as they are only used for comparison, hence the time complexity is O(1) and the space complexity is also O(1)

Original string:
'pattern$text

Reversed string:
'nrettap$txet

For each string, I gathered the list containing the z-box of each position. The use of doing this is for me to know the longest string based on the original text and reversed text that matches to its assigned pattern. I reversed the result of the z-algorithm list of the reversed string. I then create a new list to count the match of suffix and prefix in its substring. Before adding them together, I discard the first few values from the list that calculates the z-box up until the '$' symbol. This is done as I only need the z-box of the text to know the longest substring that matches the pattern, since the results of the z-box up to '$' is not needed anymore, I could just remove that. I also did this to the reversed string for the same reason.

The complexity of the z_algorithm_list is O(n+m) and the auxiliary space complexity is also O(n+m) where it stores the z value of each character from the concatenated text and pattern.

An example:

Original string

c	c	d	$	c	c	d	c	d	d
-	1	0	0	3	1	0	1	0	0

Becomes:

c	c	d	c	d	d
3	1	0	1	0	0

Reversed string

d	c	c	$	d	d	c	d	c	c
-	0	0	0	1	2	0	3	0	0

Becomes:

d	d	c	d	c	c
1	2	0	3	0	0

I then reversed the result of the sliced reverse string as I planned on making this as a longest suffix string count. In order to calculate the total matches of a substring, I add the prefix and the suffix of the same substring, therefore considering the position where there is a middle part of a substring having a transposition. The reason for iterating len(text) - len(pat) + 1 times in line 69 is because the last possible match index is in that position; if it exceeds that count, the length of the pattern has become longer than the substring left.

Example:

Reversed z-box list of reverse = 0 0 3 0 2 1

Calculate:

Original string			3	1	0	0	0	1
Reversed string	0	0	3	0	2	1		
+			6	1	2	1		

The reason why the values are counted this way is because as said before, the prefix and suffix of every substring are counted. For example, having a pattern of 3, index[0] of prefix is the same substring as index[2] of the reversed string.

The calculation that has been computed can be used to get the positions of the position of substring that matches with the pattern.

The first condition happens when the substring matches to a pattern without any transposition happening. In this case, both original string and reversed string will capture the entire length of the pattern as the z-box value.

The second condition is when there is a preposition. Since the most typographical error that can be made was between 2 successive characters, we only consider 2 characters not detected by the pattern. In this case, it is possible to eliminate all the counts that do not satisfy the rules of **len(pattern) - 2**.

To check whether the transposition is right, we can take all indexes of the list that satisfy the rules above. Then, find the index of the transposition by adding the count and the index.

Take the 2 characters starting from that index and compare it to the 2 characters of pattern starting from index count (but being reversed) in order to check whether they are the same string.

Example: (index-0 based)

Case 2a
Index 1 of count gives 1. Hence, we can check the possible transposition:
Index 1 substring = ‘cdc’
Finding transposition = start from index[1 + 1] and take the next character = ‘dc’
Find the pattern with possible transposition = index[1] and successive = ‘cd’
When the transposition pattern is reversed, it gives the same string.

Case 2b
Index 3 of count gives 1
Index 3 substring: ‘cdd’
Find transposition = start from index[3+1] and take successive = ‘dd’
Find the pattern with possible transposition gives ‘cd’
If one of the transpositions is reversed, it does not yield the same string. Hence, this will not be accepted.

The result of count can be used as an identification where transposition is, so by adding it to the current substring index, the position of transposition can be found.

The time complexity of find_pattern_allow_transposition is also O(n+m). It contains the z_algorithm_list, which is the largest complexity inside the function. There is a time complexity consumed from list slicing where it took O(m) to slice from the pattern length when getting the list of the z_algorithm. The largest auxiliary space complexity is also O(n) where the biggest complexity is from a list with a size of n taken from the return list of z_algorithm_list that has been sliced.