Preprocessing bad character

For this, a 2D list is created with a len(string) row and 26 columns, where each column represents an alphabet from a-z. A for loop is used to update the entire list of a row, based on each iteration of the index. Deepcopy is used in order to take the entire list of the previous row, but modify the rightmost position based on the current iteration. I choose to deepcopy this for several reasons. Firstly, initialising all the elements as -1 will require an unnecessary iteration when the value from the row above is modified, as this means that the same column below that rows need to be the same as that value until another same character is found in the next index. Second, shallow copy will not be possible as lists are in the form of a pointer, so modifying another row will affect the previous row and the row after, causing all lists to be the same in different rows.

There is another loop where a '.' symbol is detected. Since this only appear once, the worst complexity of preprocessing this is O((1+m)*alphabet size), which in this case is 26 and 1 additional is to modify the entire content of the list of that specific row. I used 2D list instead of only 1D in order to be able to skip more (less looping needed) when the current point that pattern is at is less than the index of the rightmost of a certain character.

Preprocess good suffix

Both good suffixes and matched prefixes were preprocessed by using the z-algorithm first. The difference between the z-algorithm is in the string_match function where the z-algorithm used by a good suffix only adds the length of the z-box when the letters of comparison are exactly the same, the wildcard '.' not being included. This is because to create a good_suffix result, we want to find the rightmost letter that matches the requirement to prevent any skips. If the wildcard is considered to be a match to anything, there might be a possibility that the algorithm thinks that '.' is the same as the previous letter before the index, hence not considered as a box, leading to a possibly bigger skip (when passed as a matched prefix).

Example: (0-indexed)
.baba at index 3 will give a value of 2, because the rightmost suffix match encountered is at 2. This is only possible as '.' and 'a' before the suffix 'ba' is considered to be different. In cases where '.' is considered.

On the other hand, when the condition is added to '.baba', the good suffix on index 3 will give -1, since the character before 'ba' was considered to be the same as the wildcard, although the wildcard can be different.

Good suffix without match '.' = [-1, -1, -1, 2, -1, 3]
Good suffix with match '.' = [-1, -1, 2, -1, 0, 3]

Preprocess matched prefix

The opposite case works for matched_prefix, the wildcard '.' should be considered a match with other characters in order for them to be included within the prefix. By not considering '.' to be the same, it is possible where a smaller prefix is found leading to a bigger skip. To find the matched prefix, the z-algorithm is used and the result of it will be used to determine the

maximum length of prefix starting from the end. This is done as matched_prefix(i) works by having a prefix from [0:certain_val] equals to substrings [i:m] where m is the last element. By working the prefix match list from the back, we can see the length of the largest z-box which determines the longest matched prefix.

Example:
'.bbbb' not allowing the wildcard to be the same as the other letter will lead to no prefix matches at all except on the starting index. This means that if there is a match or mismatch, it will skip more parts of the text that possibly could have matched.
Meanwhile, '.bbbb' that considers '.' wildcard when matching the characters will have a more accurate prefix match, which means that all the rightmost mismatch will definitely be caught and the pattern would not shift more than when wildcard is not considered.

Matched prefix without match '.' = [5, 0, 0, 0, 0, 0]
Matched prefix with match '.' = [5, 4, 3, 2, 1, 0]

Boyer Moore modifications

Most of the code has the same as the idea shown in *FIT3155_week_2_notes.pdf.* The first additional line was made during the looping to check whether character of a certain point on pattern matches the current pointed index in text. Instead of only having a requirement of the characters being the same, the current letter in the pattern can be '.' symbol too in order to be considered a match.

For Galil's optimization for matching pattern, the shift will be based on the matched_prefix of index 1. Because when match has been found, usually the next iteration will just pattern moving 1 step forward. But, the matched prefix of the next index can be used to see from that position how much the pattern can move instead of just moving by 1.

When searching for a bad char shift, I access the direct location within the table, which is taken from the current pattern index row. This is done as the rightmost of the same character can be different based on where the current point is pointed towards. Since each row in the bad character table is based on the rightmost before that certain point, accessing the row using the index immediately will find the rightmost index of pattern that matches the current bad character on text and shift to that point.

To find the good suffix, the reason I use +1 when searching by the index is because if the mismatch happens in index i, then it is impossible for the current i to be included as the good suffix. So, accessing the next index shows that the next index is part of the good index and the rightmost good suffix can be found from the good_suffix method of that index *(page 9 of FIT3155_week_2_notes)*.

Data Structure

Overall the data structure used for all is a stack. The reason why stack is preferable compared to the other data structures is mainly due to the indexing. Since most of the computation is searching, linked lists will not be a good option as each search took a time

complexity of O(m) instead of constant time. Using set is also not a good idea as it means that I will need to create extra keys in order to identify any unique alphabet and position. In cases like bad character shifts, lists work way easier than sets since only an index of the current position of the pattern and the current alphabet in text is needed to access the exact shift needed, sets will need way more information in order to create a unique key and also has a greater searching complexity. Queue is also not preferred since it does not work well in accessing a certain index to take the value, which is what is done mostly in the boyer_moore function.