A1 Question 2

Lots of things had to be done to boyer moore to make it work the other way around:

(extended) Bad character rule:

In Boyer moore, we process the bad character rule by iterating through the indices backwards, keeping a dictionary of the latest occurrence of each character and updating the bad character dictionary. The bad character dictionary is a dict that maps characters onto lists, and the lists contain at index i the leftmost occurrence of the character that isn't at index i or less.

At every index, we update the latest occurrence for the character at that index. It's quite similar to forwards preprocessing of the bad character rule but we iterate backwards instead.

Good prefix:

To preprocess the good prefix rule, we first need z_prefix. Thankfully, z_prefix is the output of the z algorithm anyway, so we can just run z algorithm on the pattern to get what we need. Whereas In the original implementation detailed in the notes, the good suffix and matched prefix tables were both size 1 more than the length of the pattern, I didn't do that for the good prefix and matched suffix arrays because I didn't see the need.

Instead of iterating forwards, we iterate backwards through the pattern. This ensures that we get the leftmost occurrence of each prefix. Whenever we encounter a Z box that's bigger than 0, then we update the index in the good_prefix array corresponding to the size of the z box with the index that it occurred. This allows us to get the index of a re-occurence of a prefix based on the size of the prefix, or check if a re-occurence exists.

Matched Suffix:

To preprocess Matched suffix, we need z_suffix. We do this by reversing the pattern, running z algorithm on it and then reversing that array.

We can then iterate through the indices, updating an index at matched suffix to a new number only if the zsuffix value there (size of the z box ending there) is one more than the index, which means that that z box covers the entire prefix from 0 to i, and also matches the suffix, which means it's a new largest matched prefix.

We then iterate through again to fill in the gaps with their previous matched suffix, which makes sense since the biggest suffix from i+1 to the end is going to be the biggest matched suffix from i to the end unless there's a new biggest suffix from i to the end.

Galil's Optimisation:

Unfortunately I don't have the time to make Galil's optimisation work, but it works much the same way as in regular boyer moore. Have a range that you've already compared, and skip it for the next iteration