Roll Number - MT16121

Name – Ankit Sharma

A. **Naïve string matching algorithm**

The text and the pattern to be matched are taken as input from the console and are converted into list of characters. Variables are initialized and a check is performed to ensure that the length of the pattern should not be greater than the length of the text against which the pattern is to be matched. If the algorithm tries to match the first character of the pattern, then we set the temp_offset to be equal to the temp_iterator (index iterator for the text) so that if the entire pattern matches then I put its value to the offset_list. Flag is set to 1 if a mismatch is detected and if it is set to 1 then we break out of the inner loop. If the last character of the pattern is matched without the flag being set to 1, then we append the value of temp_offset to the offset_list. When the iterator has reached the end of the text during comparison, then the flag is set to 2 and we break out of the outer loop as well.

The offset_list is iterated and values are printed.

For instance, if the input text is ankitankit and the pattern to be matched is ankit, then the offset_list will contain 0 and 5.

The time complexity of the algorithm is O (TP), where T is the length of the text and P is the length of the pattern.

B. **Naive string matching algorithm with one mismatch allowed**

The exact same approach is followed with the only modification to allow one mismatch. If the number of mismatch reaches 2, then we break out of the inner loop as we did in the first part.

For instance, if the input text is ankitankit and the pattern is ankiy, as we allow 1 mismatch, thereby the offset_list will contain 0 and 5.

The time complexity of the algorithm is O (TP), where T is the length of the text and P is the length of the pattern.

C. **KMP String matching algorithm**

Knuth-Morris-Pratt algorithm is a popular string matching algorithm which outperforms the naïve string matching algorithms by ensuring that the previously matched characters are not re matched. KMP algorithm can be broken down into two parts: 1. Construction of prefix table 2. Using the prefix table to perform string matching.

Step 1: Construction of the prefix table from the pattern

The size of the prefix table is same as that of the pattern. Flag has been used to mark whether last character was a match or a mismatch. If flag is reset, then the characters at i and j were checked and if they match then both the variables were incremented by 1 and flag was set to 1. If the characters at i and j do not match then, the element at the j^{th} position is set to 0 and j is incremented by 1.

If the flag is set, that is the previous check was a match, then also we check for i^{th} and j^{th} characters. If they match, then both the variables are incremented and the j^{th} element of the list is set to i + 1. In case of a mismatch, the algorithm iterates until

the elements at i and j are equal or i equals 0. Conditional statements are further used to populate the prefix table. The elements of the prefix table represent the length of the longest proper prefix in the pattern that matches a proper suffix in the same pattern.

Step 2: Using prefix table to perform string matching

The string and the pattern iterator are incremented till the corresponding text and pattern characters are equal. If the last character of the pattern matches to that of the last character of the text, offset_list is updated and the pattern iterator is updated. Otherwise, if the pattern_iterator is non zero, then it is updated using the prefix table or else string iterator is incremented.

For instance, if the input text is aaaaaa and the pattern to be matched is aa, then the offset_list will contain 0, 1, 2, 3 and 4. The prefix table will contain 0 and 1.

The time complexity of the algorithm is O (T + P), where T is the length of the text and P is the length of the pattern.