Report a1q3

Main Idea

To make pattern matching work with special characters like #, I extended the standard Burrows-Wheeler Transform (BWT) search method. Normally, BWT uses something called backward search to find exact matches. This means we look at the pattern from the end (right side) and move backwards, narrowing down the possible places it could appear in the text. When the pattern contains a wildcard like "#", the search process becomes more flexible, instead of matching a specific character, the algorithm branches out and tries every possible character that appears in the BWT, except the special end-of-text symbol $. In this implementation, the BWT is constructed using the naive method which is by generating all cyclic rotations of the text, sorting them lexicographically, and extracting the last column. This also produces the suffix array, which maps each row of the BWT back to its starting position in the original text. During pattern matching, the algorithm performs a backward search through the BWT, moving from the end of the pattern to the beginning. At each position where there is a '#', it explores all possible character matches by branching into multiple search paths, each tracked independently to ensure no valid match is missed. Once the entire pattern has been processed, the suffix array is used to recover the original starting positions of all matches in the text. These positions are then converted to 1-based indexing and written to the output, including all cases where wildcards matched different characters.

For the worst case time complexity, we know that the naive BWT construction takes O(n^2 log n) since we generate all n cyclic rotations, each of length n which is O(n^2) and we sort them (n log n number of comparisons and O(n) cost per comparisons since string length n) so we get O(n^2 log n). When creating the rank array, we get O(|a| * n) where |a| is the size of the characters. The backward search function begins with a single search range over the entire BWT, which takes constant time O(1). As it processes the pattern from right to left, each character either narrows the range using precomputed rank arrays and first occurrence tables (for regular characters), or causes branching over all possible characters in the BWT (for wildcards), excluding the separator character '$'. If the pattern contains k wildcards, the number of search paths can grow exponentially up to |a|^k, where |a| is the alphabet size. Each path may process up to m characters and collect up to n matches if the range spans the full BWT where m is the length of the pattern and n is the length of the string. Therefore, the worst-case time complexity is O(|a|^k · m · n).

As for the worst case space complexity, the BWT and suffix array both takes O(n) space and to get the cyclic rotations, we need n rotations stored, each of length n, so O(n^2). The rank array takes O(|a| * n) since it stores n+1 size array for each character. The first occurrence table uses O(|a|) at most. Finally, the ranges stack uses O(|a|^k) where k is the number of wildcards and matches uses at most O(n) space to store the positions. The total worst case space complexity is O(n^2 + |a| × n + |a|^k)