Algorithm 1 : naive string matching
The algorithm takes a flag variable initially 1 which implies that the string is a match, performing the linear search for pattern in text we put flag = 0 as soon as we find a mismatch. Else if it's a match we output the corresponding index at which we found a match and continue further.

Suppose the length of text is : n
And the length of pattern is : m

The time complexity of the algorithm is O(nm):
There are two loops here: the first one runs n times for each character of text and the inner loop runs m times for the pattern matching. Thus the total no. of times instruction runs will be n*m.

Example : text = 'TGAGAGAC' and pattern = 'GAG'
Here we start with T and match it with G. We find a mismatch and we break out the inner loop. W iterate to G and match with pattern[0] i.e., G; it's a match and thus we go on further to A matching text[i+1] with pattern[1]. Finally we get a complete match here and thus we print the value of i (the index)

output : occurrence at 1,3

Algorithm 2: string matching with one string mismatch allowed
The algorithm is same as algorithm 1 with a count initialised to 0 at each iteration which is increased as soon as we find a mismatch. Later on we check if flag=1 only then we consider that the string matched and we then print the corresponding index of the text where the match occurred.

Suppose the length of text is : n
And the length of pattern is : m

The time complexity of the algorithm is O(nm):
There are two loops here: the first one runs n times for each character of text and the inner loop runs m times for the pattern matching. Thus the total no. of times instruction runs will be n*m.

Example : text = 'TGAGAGAC' and pattern = 'GAG'
Here we start with T and match it with G. We find a mismatch and we break out the inner loop. Lets consider for index 5 where intially the count is 0.we find a match at G and A thus count remains 0 but increases at C which is a mismatch from the pattern[2]. Now count ==1.But the count here is also not greater than 1. Thus we dont break out of the loop and later check flag which is 1 and we say its a match.

output : occurrence at 1, 3, 5

Algorithm 3: KMP string matching algorithm

The naive searching algorithm doesn't work well in cases where we see matching characters followed by a mismatching character.The basic idea behind KMP's algorithm is: whenever we detect a mismatch (after some matches), we already know some of the characters in the text of next window. We take advantage of this information to avoid matching the characters that we know will anyway match.It checks in the pattern if we have same suffix and prefix when there is a mismatch,and utilises this information to avoid further check in text. Thus this algorithm takes lesser time than naive search algorithm.

KMP algorithm works with a time complexity of O(n+m) where n is the length of text and m is the length of pattern.

Example : generation of pat list for pattern "ABXAB" is as follows :
We take an empty list pat and initialise its 0th element as 0 and take j at position 0 and i at position 1.on matching character at i and j in pattern we find A and B are not same thus we increment i. .again A and X are a mismatch and we move i to A. here we find a match and put in pat[i] = j+1 i.e., 1 here in this case and increment i and j. Now we match B and B. we find match and put pat[4] = 1+1=2. Thus the list pat now contains [0,0,0,1,2] which shows that we have same suffix and prefix AB.
Now we use this prefix calculation in our KMPSearch function. We move forward as we did in naive approach but when we find a mismatch, we check if we had same prefix as in pat by the value 1 in pat matrix and thus don't check the string from the next charcter of text as in naive but from the character where we found a mismatch. Which reduces the characters to be compared by a huge amount. Thus KMP is more efficient that naive algorithm.