Bharatiya Vidya Bhavan's
SARDAR PATEL INSTITUTE OF TECHNOLOGY
(Autonomous Institute Affiliated to University of Mumbai)
Munshi Nagar, Andheri (W), Mumbai – 400 058.
Department of Computer Engineering

Experiment	9
Aim	To understand and implement String Matching Algorithm
Objective	1) Write Pseudocode for any 2 string matching algorithm 2) Implementing the above mentioned 2 string matching algorithm 3) Calculating time complexity of the given problems 4) Solve the string matching for both the algorithm on pen and paper
Name	Atharva Pramod Patil
UCID	2022600040
Class	Ai-Ml
Batch	C
Date of Submission	16 April

Algorithm and Explanation of the technique used

Naive Method:

pseudo code:

```
procedure search(pat, txt):
    M := length of pat
    N := length of txt

    for i := 0 to (N - M):
        j := 0

        while j < M and txt[i + j] = pat[j]:
            j := j + 1

        if j equals M:
                print "Pattern found at index",
```

1. Loop through the text from index 0 to (N - M), where N is the length of the text and M is the length of the pattern.
2. For each index i, initialize a variable j to 0.
3. Within a nested loop, compare each character of the pattern with the corresponding character in the text starting from index i.
4. If all characters of the pattern match the corresponding characters in the text starting from

index i, print the index i where the pattern is found.

5. Repeat steps 2-4 until all possible starting indices for the pattern in the text have been checked.

KMP Method
psuedo code:

```
procedure search(pat, txt):
    M := length of pat
    N := length of txt
    sub := an array of length M to store the prefix-suffix overlap lengths for the pattern

    // Preprocess the pattern to generate the prefix-suffix overlap lengths
    prefixSuffix(sub, pat, M)

    i := 0  // Index for the text
    j := 0  // Index for the pattern

    while i < N:
        if pat[j] equals txt[i]:
            i := i + 1
            j := j + 1

            if j equals M:
                print "Pattern found at index", (i - j)
                j := sub[j - 1]  // Move j to the next possible match position based on prefix-suffix overlap
        else:
            if j is not equal to 0:
                j := sub[j - 1]  // Move j to the next possible match position based on prefix-suffix overlap
            else:
                i := i + 1

procedure prefixSuffix(sub, pat, M):
    len := 0
    sub[0] := 0  // Initialize the first element of the sub array

    for i := 1 to (M - 1):
        if pat[i] equals pat[len]:
            len := len + 1
            sub[i] := len
        else:
            if len is not equal to 0:
```

	len := sub[len - 1] // Adjust len based on the previous prefix-suffix overlap i := i - 1 // Stay at the current index to recheck with the adjusted len else: sub[i] := 0 1. Preprocess the pattern to generate the prefix-suffix overlap lengths using the `prefixSuffix` procedure. 2. Initialize indices `i` and `j` for the text and pattern, respectively. 3. Compare each character of the text with the corresponding character of the pattern. 4. If a mismatch occurs, update the index `j` based on the prefix-suffix overlap information stored in the `sub` array. 5. Repeat steps 3-4 until either the pattern is found in the text or the end of the text is reached.

Program(Code)	**Naive Method:**
	`def pattern(txt ,pat):`
	`M = len(txt)`
	`N = len(pat)`
	
	`for i in range (M-N+1):`
	`j = 0`
	
	`while(j<N):`
	`if(txt[i+j]!= pat[j]):`
	`break`
	`j += 1`
	
	`if(j == N):`
	`print("Pattern at index: ",i+1)`
	
	`if __name__ == '__main__':`
	`txt = 'ABCDADDSCABCDADBCABCAAD'`
	`pat = 'ABC'`
	
	`pattern(txt ,pat)`
	
	Kmp Method:
	`def pattern(txt, pat):`
	`M = len(txt)`

	<pre>N = len(pat)
	sub = [0]*N
	j = 0
	
	prefixSuffix(sub, pat, N)
	i = 0
	while (M - i) >= (N - j):
	if pat[j] == txt[i]:
	i += 1
	j += 1
	if j == N:
	print("Pattern found at index:", i - j+1)
	j = sub[j - 1]
	elif i < M and pat[j] != txt[i]:
	if j != 0:
	j = sub[j - 1]
	else:
	i += 1
	
	def prefixSuffix(sub, pat, N):
	length = 0
	sub[0] = 0
	i = 1
	while i < N:
	if pat[i] == pat[length]:
	length += 1
	sub[i] = length
	i += 1
	else:
	if length != 0:
	length = sub[length - 1]
	else:
	sub[i] = 0
	i += 1
	
	if __name__ == '__main__':
	txt = "ABCBABCDABCDCABCBABCBBAABCBABC"
	pat = "ABCBABC"
	pattern(txt, pat)</pre>

Output	Naive Method:

	`students@students-HP-280-G3-SFF-Business-PC:~/Desktop$ python3 naive.py`
	`Pattern at index:  1`
	`Pattern at index:  10`
	`Pattern at index:  18`
	Kmp Method:
	`students@students-HP-280-G3-SFF-Business-PC:~/Desktop$ python3 kmp.py`
	`Pattern found at index: 1`
	`Pattern found at index: 14`
	`Pattern found at index: 24`

Justification of the complexity calculated	**Naive Method:** Best Case Scenario: For the best case scenario we hope the pattern is found at the first position itslef ,therefore there are just N(lenght of the pattern) comparisons made. Therefore for the best Case Scenario the time complexity is O(n) Worst Case Scenario: For the worst case ,we assume that the pattern doesnt appear anywhere except for the end of the string. Here the algorith will peform (n-m+1)*m itterations for every single element in the string ,where m is the lenght of the string and n is the length of the pattern. Therefore the timecomplexity being O((n-m+1)*m). **KMP Method:** Best Case: O(n + m) In the best-case scenario, when the pattern is found at the beginning of the text or early on, the KMP algorithm typically performs a constant number of comparisons. Worst Case: O(n + m) In the worst-case scenario, where` the pattern doesn’t appear in the text at all or appears only at the very end, the KMP algorithm still maintains a time complexity of O(n + m). This is because it optimally utilizes the knowledge of the pattern's prefix-suffix overlap to avoid redundant comparisons, resulting in linear time complexity relative to the size of both the text and the pattern.
Conclusion	Through the above two codes i understood how string matching algorithms work. I used two algorithms ,1.Naive 2.Kmp method. Out of which i realised the KMP is a much better appraoch than naive caus it reduces the common patterns the the comparison string ,subsequencyly reducing the time complexity for it. I can now successfully implement both the algortihms for string comparison.