Name: Chirag Rana
Class: SE Comps
Batch: C
UID: 2018130043

EXPERIMENT – 8

AIM: Implement the KMP algorithm for string Matching

THEORY:

The Knuth-Morris-Pratt (KMP)Algorithm

Knuth-Morris and Pratt introduce a linear time algorithm for the string matching problem. The worst case complexity of the Naive algorithm is O(m(n-m+1)).A matching time of O (n) is achieved by avoiding comparison with an element of 'S' that have previously been involved in comparison with some element of the pattern 'p' to be matched. i.e., backtracking on the string 'S' never occurs.

Components of KMP Algorithm:

1. The Prefix Function (Π): The Prefix Function, Π for a pattern encapsulates knowledge about how the pattern matches against the shift of itself. This information can be used to avoid a useless shift of the pattern 'p.' In other words, this enables avoiding backtracking of the string 'S.'

2. The KMP Matcher: With string 'S,' pattern 'p' and prefix function 'Π' as inputs, find the occurrence of 'p' in 'S' and returns the number of shifts of 'p' after which occurrences are found.

Unlike Naive algorithm, where we slide the pattern by one and compare all characters at each shift, we use a value from lps[] to decide the next characters to be matched. The idea is to not match a character that we know will anyway match.
How to use lps[] to decide next positions (or to know a number of characters to be skipped)?

- We start comparison of pat[j] with j = 0 with characters of current window of text.
- We keep matching characters txt[i] and pat[j] and keep incrementing i and j while pat[j] and txt[i] keep **matching**.
- When we see a **mismatch**
  - We know that characters pat[0..j-1] match with txt[i-j…i-1] (Note that j starts with 0 and increment it only when there is a match).
  - We also know (from above definition) that lps[j-1] is count of characters of pat[0…j-1] that are both proper prefix and suffix.

- From above two points, we can conclude that we do not need to match these lps[j-1] characters with txt[i-j…i-1] because we know that these characters will anyway match. Let us consider above example to understand this.

ALGORITHM:

COMPUTE- PREFIX- FUNCTION (P)

1. m ←length [P] //'p' pattern to be matched
2. Π [1] ← 0
3. k ← 0
4. for q ← 2 to m
5. do while k > 0 and P [k + 1] ≠ P [q]
6. do k ← Π [k]
7. If P [k + 1] = P [q]
8. then k← k + 1
9. Π [q] ← k
10. Return Π

KMP-MATCHER (T, P)

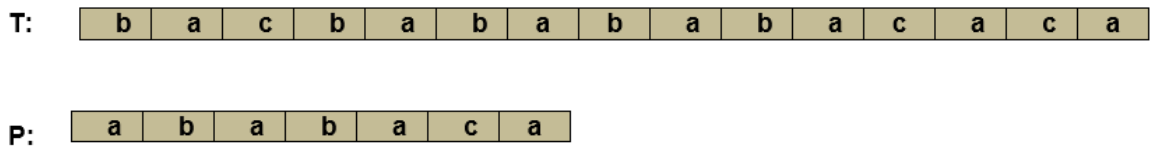
1. n ← length [T]
2. m ← length [P]
3. Π← COMPUTE-PREFIX-FUNCTION (P)
4. q ← 0 // numbers of characters matched
5. for i ← 1 to n // scan S from left to right
6. do while q > 0 and P [q + 1] ≠ T [i]
7. do q ← Π [q] // next character does not match
8. If P [q + 1] = T [i]
9. then q ← q + 1 // next character matches
10. If q = m // is all of p matched?
11. then print "Pattern occurs with shift" i - m
12. q ← Π [q]

TIME COMPLEXITY:

The for loop from step 4 to step 10 runs 'm' times. Step1 to Step3 take constant time. Hence the running time of computing prefix function is O (m).
The for loop beginning in step 5 runs 'n' times, i.e., as long as the length of the string 'S.' Since step 1 to step 4 take constant times, the running time is dominated by this for the loop. Thus running time of the matching function is O (n).

DESIGN AND IMPLEMENTATION:

Example: Compute LCS table for the pattern 'p' below:



Solution:

Initially: m = length [p] = 7

Π [1] = 0

k = 0

Step 1: q = 2, k = 0

Π [2] = 0

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0					

Step 2: q = 3, k = 0

Π [3] = 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1				

Step3: q =4, k =1

Π [4] = 2

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	A
π	0	0	1	2			

Step4: q = 5, k =2

Π [5] = 3

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3		

Step5: q = 6, k = 3

Π [6] = 0

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3	0	

Step6: q = 7, k = 1

Π [7] = 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3	0	1

After iteration 6 times, the prefix function computation is complete:

q	1	2	3	4	5	6	7
p	a	b	A	b	a	c	a
π	0	0	1	2	3	0	1

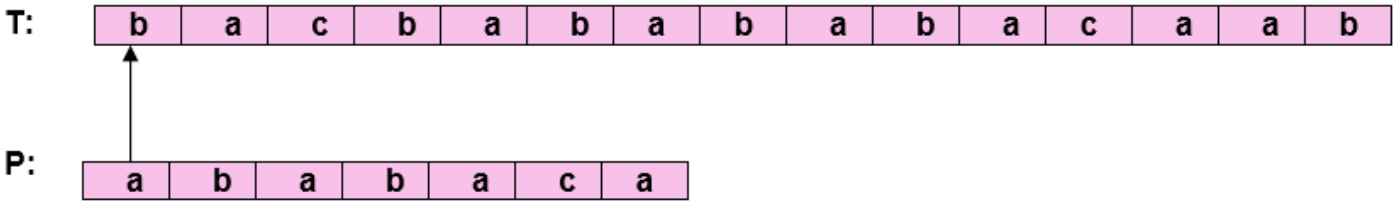
FOR KMP SEARCH:

Initially: n = size of T = 15

m = size of P = 7

Step1: i=1, q=0

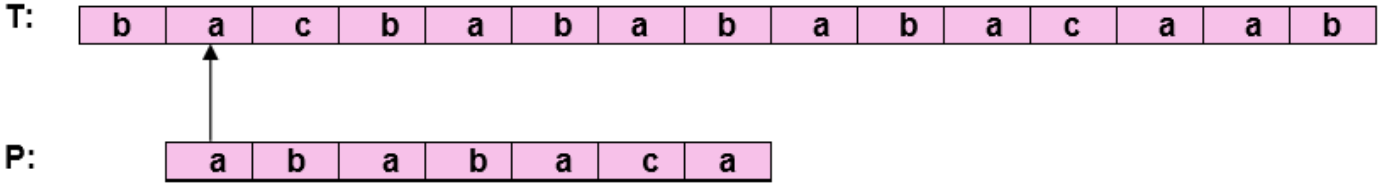
Comparing P [1] with T [1]



P [1] does not match with T [1]. 'p' will be shifted one position to the right.

Step2: i = 2, q = 0

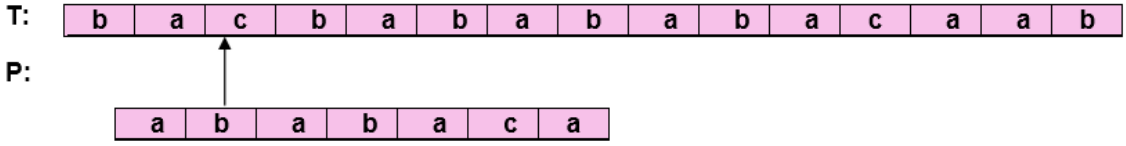
Comparing P [1] with T [2]



P [1] matches T [2]. Since there is a match, p is not shifted.

Step 3: i = 3, q = 1

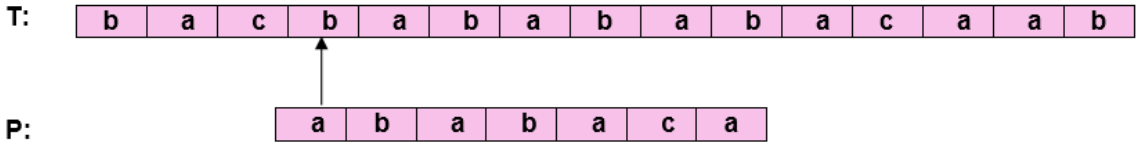
Comparing P [2] with T [3] P [2] doesn't match with T [3]



Backtracking on p, Comparing P [1] and T [3]

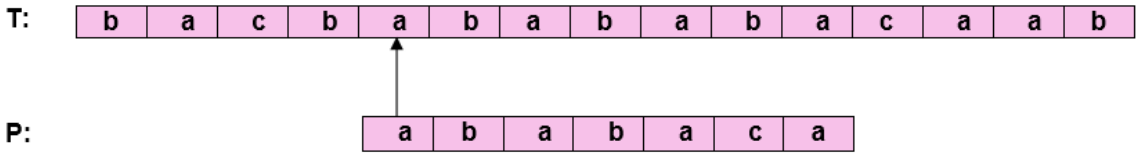
Step4: i = 4, q = 0

Comparing P [1] with T [4] P [1] doesn't match with T [4]



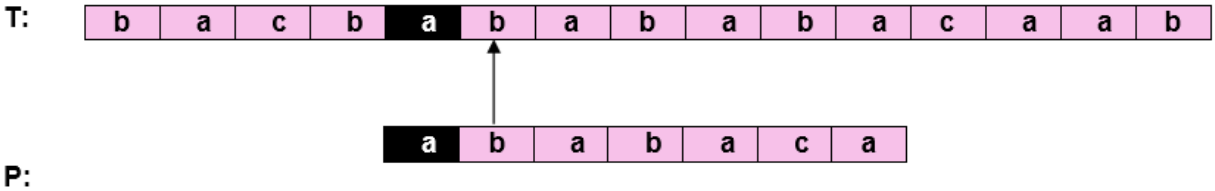
Step5: i = 5, q = 0

Comparing P [1] with T [5] P [1] match with T [5]



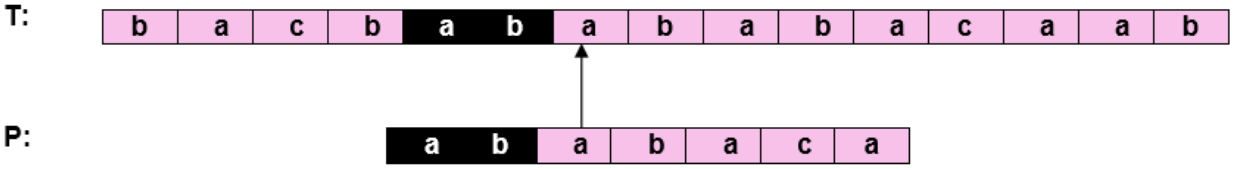
Step6: i = 6, q = 1

Comparing P [2] with T [6] P [2] matches with T [6]



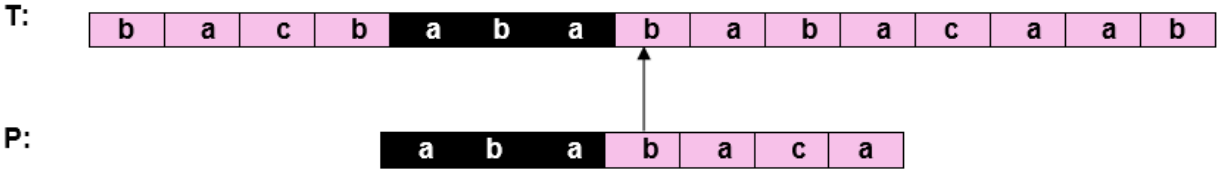
Step7: i = 7, q = 2

Comparing P [3] with T [7] P [3] matches with T [7]



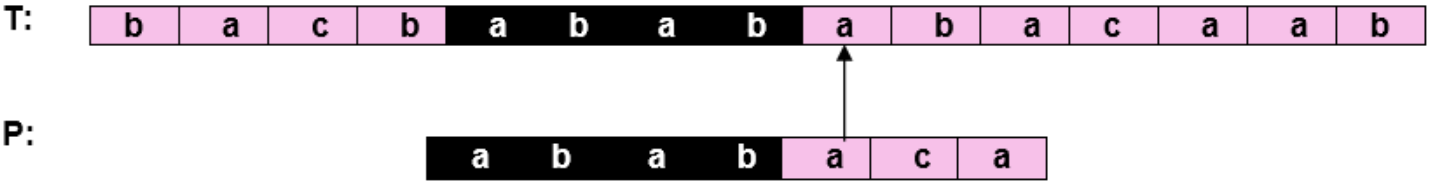
Step8: i = 8, q =3

Comparing P [4] with T [8] P [4] matches with T [8]



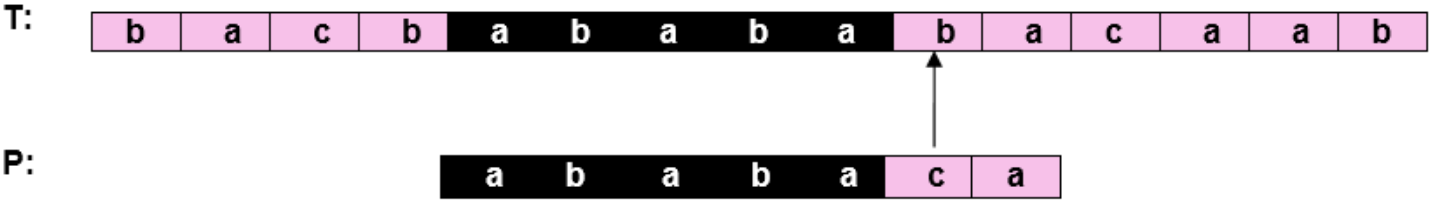
Step9: i = 9, q = 4

Comparing P [5] with T [9] P [5] matches with T [9]



Step10: i = 10, q = 5

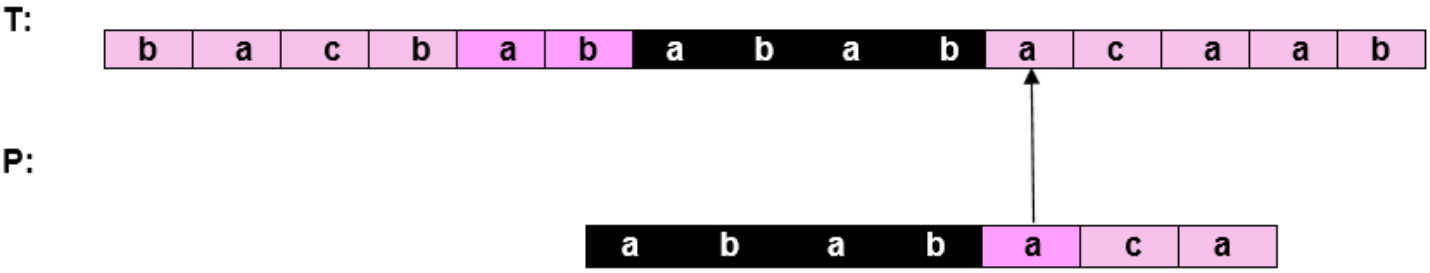
Comparing P [6] with T [10] P [6] doesn't match with T [10]



Backtracking on p, Comparing P [4] with T [10] because after mismatch q = π [5] = 3

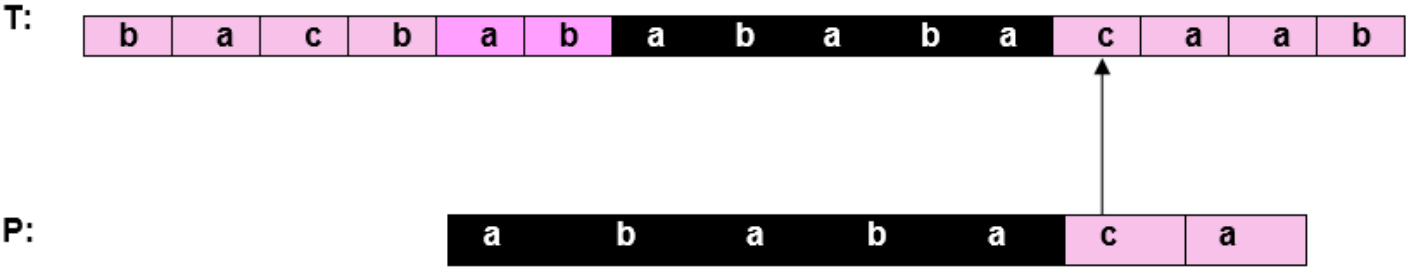
Step11: i = 11, q =4

Comparing P [5] with T [11] P [5] match with T [11]

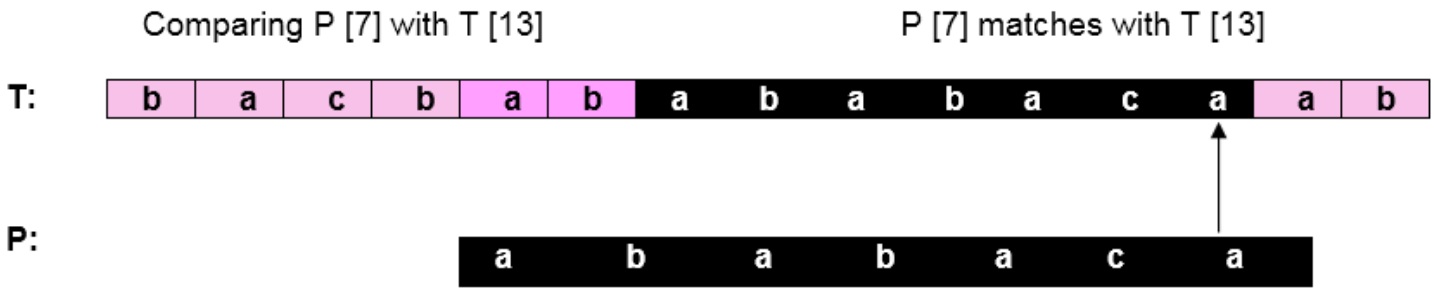


Step12: i = 12, q = 5

Comparing P [6] with T [12] P [6] matches with T [12]



Step13: i = 3, q = 6



Pattern 'P' has been found to complexity occur in a string 'T.' The total number of shifts that took place for the match to be found is i-m = 13 - 7 = 6 shifts.

CODE:

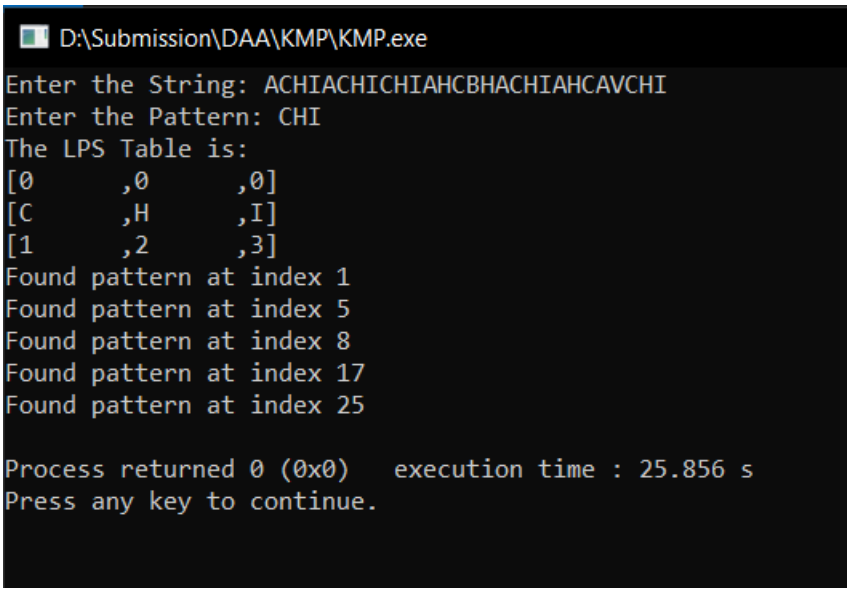
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void computeLPSArray(char* pattern, intpatternlen, int* lps);

void KMP(char* pattern, char* string)
{
	int patternlen = strlen(pattern);
	int stringlen = strlen(string);
	int lps[patternlen];

	computeLPSArray(pattern, patternlen, lps);

	printf("The LPS Table is: \n");
	printf("[");
	for (int i = 0; i < patternlen ; i++){
		printf("%d", lps[i]);
		if( i != patternlen-1){
			printf(" \t,");
		}else{
			printf("]\n");
		}
	}
	printf("[");
```

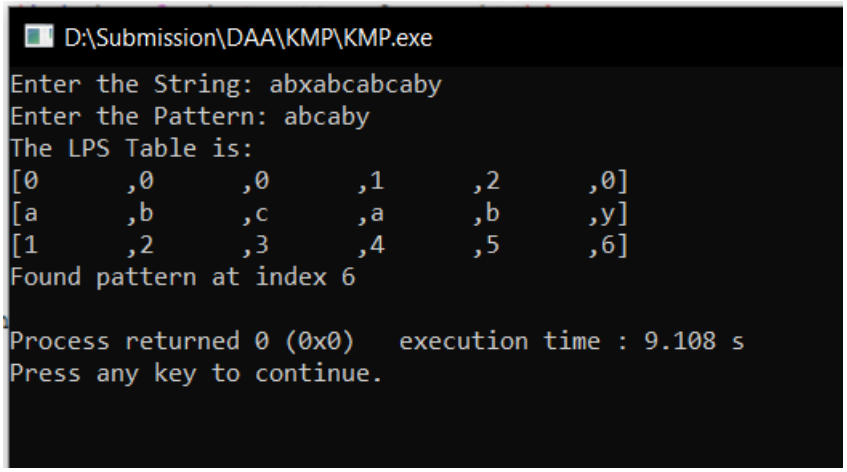
```
for (int i = 0; i < patternlen ; i++){
        printf("%c", pattern[i]);
        if( i != patternlen-1){
                printf(" \t,");
        }else{
                printf("]\n");
        }
}

printf("[");
for (int i = 1; i < patternlen+1 ; i++){
        printf("%d",i);
        if( i != patternlen){
                printf(" \t,");
        }else{
                printf("]\n");
        }
}

int i = 0;
int j = 0;
while (i < stringlen ) {
        if (pattern[j] == string[i]) {
                j++;
                i++;
        }

        if (j == patternlen) {
                printf("Found pattern at index %d \n", i - j);
                j = lps[j - 1];
        }

        else if (i < stringlen && pattern[j] != string[i]) {
                        if (j != 0)
                        j = lps[j - 1];
                else
                        i = i + 1;
        }
```

```
        }
}

void computeLPSArray(char* pattern, int patternlen, int* lps)
{
        int len = 0;
        lps[0] = 0;
        int i = 1;
        while (i < patternlen) {
                if (pattern[i] == pattern[len]) {
                        len++;
                        lps[i] = len;
                        i++;
                }
                else
                {
                        if (len != 0) {
                                len = lps[len - 1];
                        }
                        else
                        {
                                lps[i] = 0;
                                i++;
                        }
                }
        }
}

int main()
{
    char pattern[100];
    char string[100];
        printf("Enter the String: ");
    gets(string);
        printf("Enter the Pattern: ");
    gets(pattern);
        KMP(pattern, string);
        return 0;
```

}

OUTPUT:





CONCLUSION:

1. I implemented Knuth-Morris-Pratt to search for a pattern.
2. I understood the how this algorithm searches the pattern in O(n) complexity.