Knuth Morris Pratt's (KMP) String Matching Algorithm

Presented By:

1. Tasneem Rahman Adiba (011191259)
2. Debopom Sutradhar (011201046)
3. Mahbub Hasan (011181134)
4. Yeasir Arafat (011201035)
5. Sami Nayeem (011193020)

Table of contents

01

Concept/Motivation

Why should we use this algorithm?

02

Simulation

How this algorithm works?

03

Pseudocode

What can be the procedure?

04

Code

The applied C++ code for this algorithm

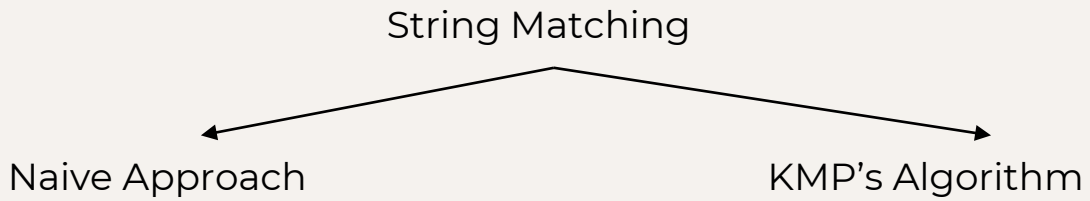
05

Time Complexity

The runtime analysis

Concept

The problem of finding occurrence(s) of a pattern string within another string or body of text . There are many different algorithms for efficient searching. Also known as exact string matching, string searching, text searching



Naive Approach

The naive algorithm finds all the valid shifts using a loop that checks the condition **P[1.....m] = T[s + 1.....s + m]** for each of the **n-m+1** possible values of **s**

KMP’s Algorithm

Knuth Morris Pratt proposed the first time complexity algorithm for the string matching problem

The time complexity of KMP’s algorithm is O(m+n). Meanwhile the time complexity naive approach is O(m*n)

KMP’s Algorithm

The time complexity of KMP’s algorithm is achieved by avoiding comparisons with elements 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

Functions of KMP’s Algorithm

There are 2 major functions of KMP’s algorithm which are-

1. The prefix function, Pl

1. The KMP Matcher

The prefix function

The prefix function or PI function for a pattern encapsulates knowledge about how the pattern matches against shifts of itself. This information can be used to avoid useless shifts of the pattern ‘p’. In other words this enables avoiding backtracking on the string ‘S’.

The KMP Matcher

Finds the occurrence of 'p' in 'S' and returns the number of shifts of 'p' after which the occurrence is detected, using string 'S', pattern 'p', and prefix function 'Pl' as inputs.

PI Table or LPS

a c b a c b a

1	2	3	4	5	6	7
a	c	b	a	c	b	a

PI Table or LPS

a c b a c b a

1	2	3	4	5	6	7
a	c	b	a	c	b	a
0						

PI Table or LPS

a c b a c b a

1	2	3	4	5	6	7
a	c	b	a	c	b	a
0	0					

PI Table or LPS

a c b a c b a

1	2	3	4	5	6	7
a	c	b	a	c	b	a
0	0	0				

PI Table or LPS

a c b a c b a

1	2	3	4	5	6	7
a	c	b	a	c	b	a
0	0	0	1			

PI Table or LPS

a c b a c b a

1	2	3	4	5	6	7
a	c	b	a	c	b	a
0	0	0	1	2		

PI Table or LPS

a c b a c b a

1	2	3	4	5	6	7
a	c	b	a	c	b	a
0	0	0	1	2	3	

PI Table or LPS

a c b a c b a

1	2	3	4	5	6	7
a	c	b	a	c	b	a
0	0	0	1	2	3	1

PI Table or LPS

String : a q a c b r a c b a c b a

i

j

0 1 2 3 4 5 6 7

Pattern : a c b a c b a

1	2	3	4	5	6	7
a	c	b	a	c	b	a
0	0	0	1	2	3	1

PI Table or LPS

String : a q a c b r a c b a c b a

i

j

0 1 2 3 4 5 6 7

Pattern : a c b a c b a

1	2	3	4	5	6	7
a	c	b	a	c	b	a
0	0	0	1	2	3	1

PI Table or LPS

String : a q a c b r a c b a c b a

i

j

0 1 2 3 4 5 6 7

Pattern : a c b a c b a

1	2	3	4	5	6	7
a	c	b	a	c	b	a
0	0	0	1	2	3	1

PI Table or LPS

String : a q a c b r a c b a c b a

i

j

0 1 2 3 4 5 6 7

Pattern : a c b a c b a

1	2	3	4	5	6	7
a	c	b	a	c	b	a
0	0	0	1	2	3	1

PI Table or LPS

String : a q a c b r a c b a c b a

i

j

0 1 2 3 4 5 6 7

Pattern : a c b a c b a

1	2	3	4	5	6	7
a	c	b	a	c	b	a
0	0	0	1	2	3	1

PI Table or LPS

String : a q a c b r a c b a c b a

i

j

0 1 2 3 4 5 6 7

Pattern : a c b a c b a

1	2	3	4	5	6	7
a	c	b	a	c	b	a
0	0	0	1	2	3	1

PI Table or LPS

i

String : a q a c b r a c b a c b a

j

0 1 2 3 4 5 6 7

Pattern : a c b a c b a

1	2	3	4	5	6	7
a	c	b	a	c	b	a
0	0	0	1	2	3	1

PI Table or LPS

String : a q a c b r a c b a c b a

i

j

0 1 2 3 4 5 6 7 8

Pattern : a c b a c b a

1	2	3	4	5	6	7
a	c	b	a	c	b	a
0	0	0	1	2	3	1

PI Table or LPS

String : a q a c b r a c b a c b a

i

j

0 1 2 3 4 5 6 7

Pattern : a c b a c b a

1	2	3	4	5	6	7
a	c	b	a	c	b	a
0	0	0	1	2	3	1

PI Table or LPS

String : a q a c b r a c b a c b a

i

j

0 1 2 3 4 5 6 7

Pattern : a c b a c b a

1	2	3	4	5	6	7
a	c	b	a	c	b	a
0	0	0	1	2	3	1

PI Table or LPS

String : a q a c b r a c b a c b a

i

j

0 1 2 3 4 5 6 7

Pattern : a c b a c b a

1	2	3	4	5	6	7
a	c	b	a	c	b	a
0	0	0	1	2	3	1

PI Table or LPS

String : a q a c b r a c b a c b a

i

j

0 1 2 3 4 5 6 7

Pattern : a c b a c b a

1	2	3	4	5	6	7
a	c	b	a	c	b	a
0	0	0	1	2	3	1

Pseudocode (Prefix Function)

```
function calc_ps(pattern,p_len)
    lps[p_len];
    lps[0]=0;
    length = 0;
    for i=1 to p_len
        if pattern[i] == pattern[length]
            lps[i] = length + 1;
            length++; i++;
        else
            if length != 0
                length = lps[length - 1];
            else
                lps[i] = length;
                i++;
    return lps;
End
```

Pseudocode (KMP Matcher)

```
function kmp (str, pattern, p_len, s_len, & foundAt)
   lps[] = calc_ps(pattern,p_len,s_len);
   i = 0, j = 0;
   while (i < s_len)
      if str[i] == pattern[j]
         i++; j++;
      else
         if (j != 0) j = lps[j - 1];
         else i++;

      if j == p_len
         foundAt = i-p_len;
         j = lps[j-1];
         return true;
   return false;
End
```

Code for KMP

```
int main() {
    string str="aqacbracbacba";
    string pattern="acbacba";
    int p_len = pattern.length();
    int s_len = str.length();
    int foundAt;
    bool isFound = kmp(str, pattern,p_len,s_len,foundAt);
    if(isFound){
        cout<<"Found at index "<<foundAt<<" to "<<foundAt+p_len-1;
    }
    else{
        cout<<"Not Found";
    }

    return 0;
}
```

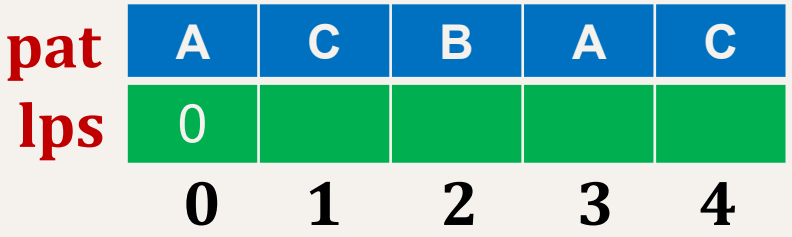
Code for KMP

```cpp
#include <bits/stdc++.h>
using namespace std;

vector<int> calc_ps(string pattern,int p_len) {
    vector<int> lps(p_len);
    int length = 0;
    int i = 1;
    while (i < p_len) {
        if (pattern[i] == pattern[length]){
            lps[i] = length + 1;
            length++; i++;
        }
        else {
            if (length != 0)
                length = lps[length - 1];
            else{
                lps[i] = length;
                i++;
            }
        }
    }
    return lps;
}
```

```cpp
bool kmp (string str, string pattern,int p_len,int
s_len,int& foundAt) {

    vector<int> lps = calc_ps(pattern,p_len);
    int i = 0, j = 0;

    while (i < s_len) {
        if (str[i] == pattern[j]){
            i++; j++;
        }
        else {
            if (j != 0)
                j = lps[j - 1];
            else
                i++;
        }

        if (j == p_len) {
            foundAt = i-p_len;
            j = lps[j-1];
            return true;
        }
    }
    return false;
}
```

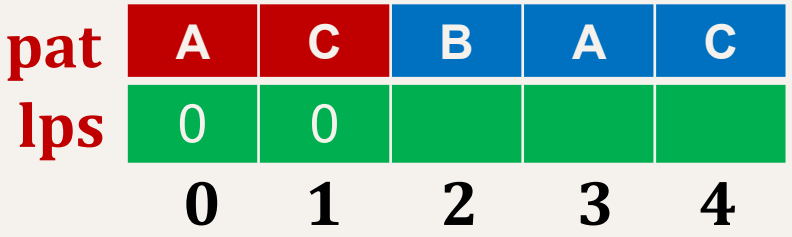
Pattern: ACBAC

length = 0 i=0



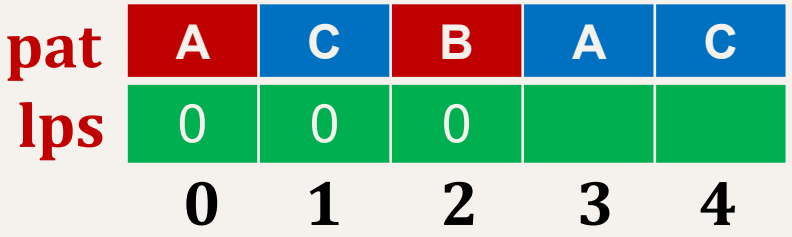
Pattern: ACBAC

length = 0 i=1



Pattern: ACBAC

length = 0 i=2



Pattern: ACBAC

length = ~~0~~ 1 i=3

pat	A	C	B	A	C
lps	0	0	0	1	
	0	1	2	3	4

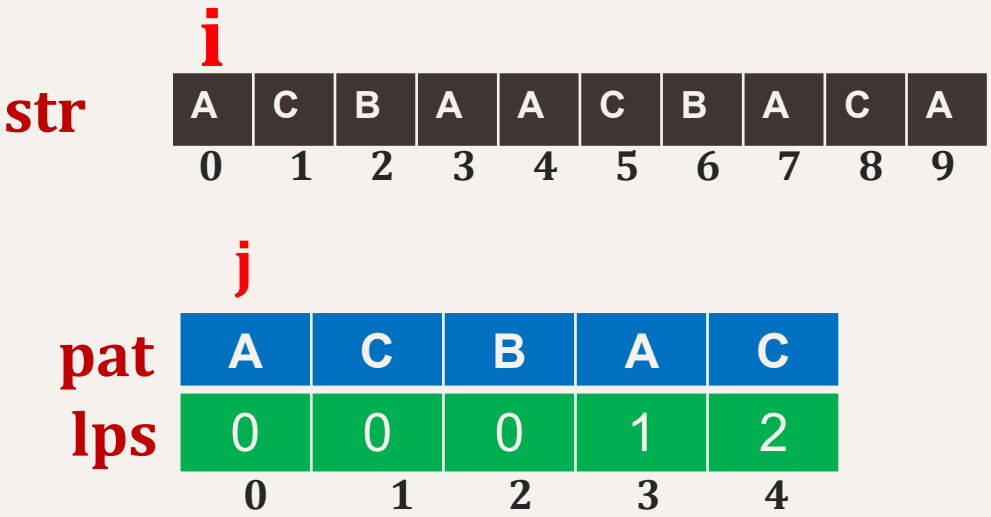
Pattern: ACBAC

length = ~~1~~ 2 i=~~4~~

pat	A	C	B	A	C
lps	0	0	0	1	2
	0	1	2	3	4

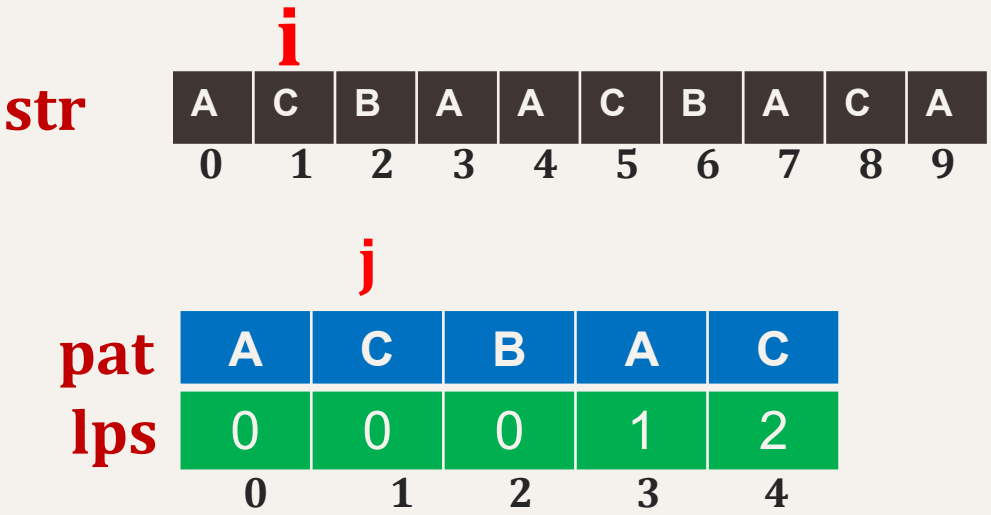
String: ACBAACBACA

Pattern: ACBAC



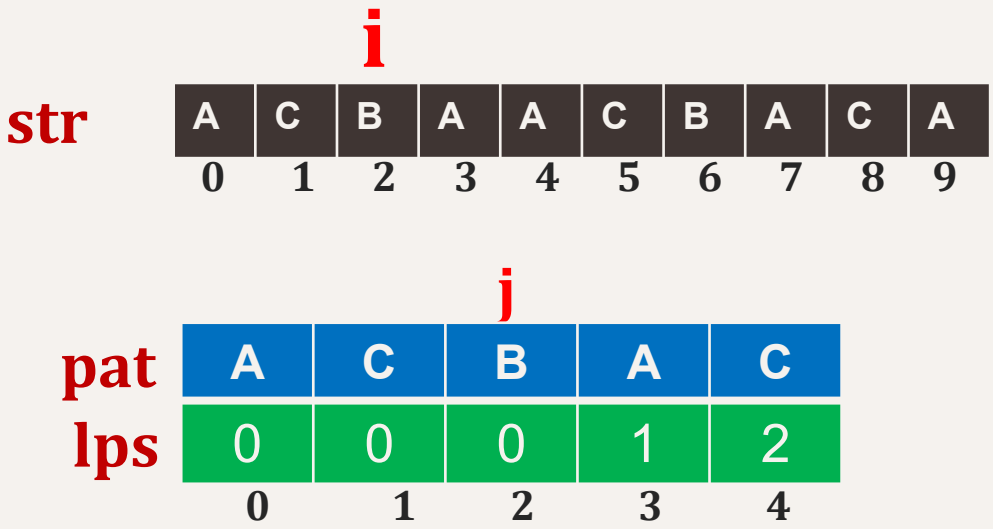
String: ACBAACBACA

Pattern: ACBAC



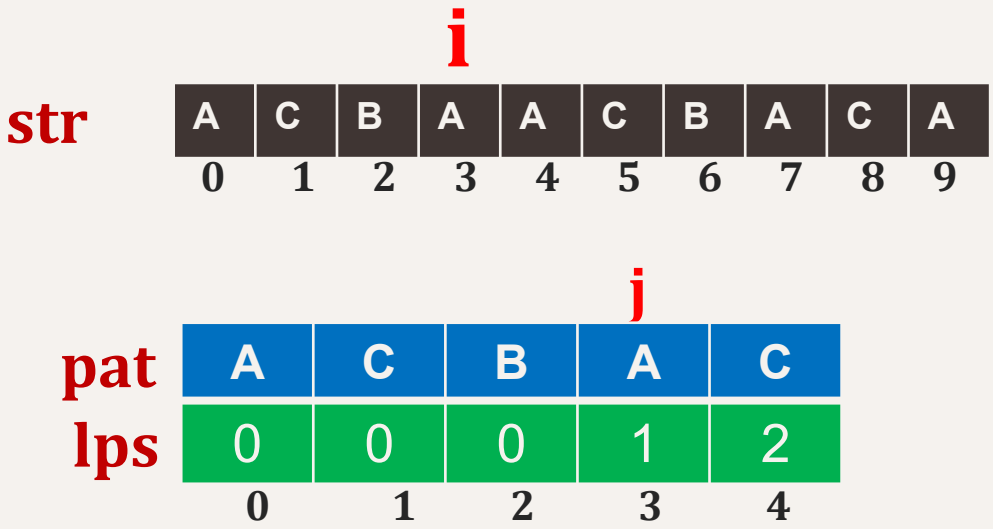
String: ACBAACBACA

Pattern: ACBAC



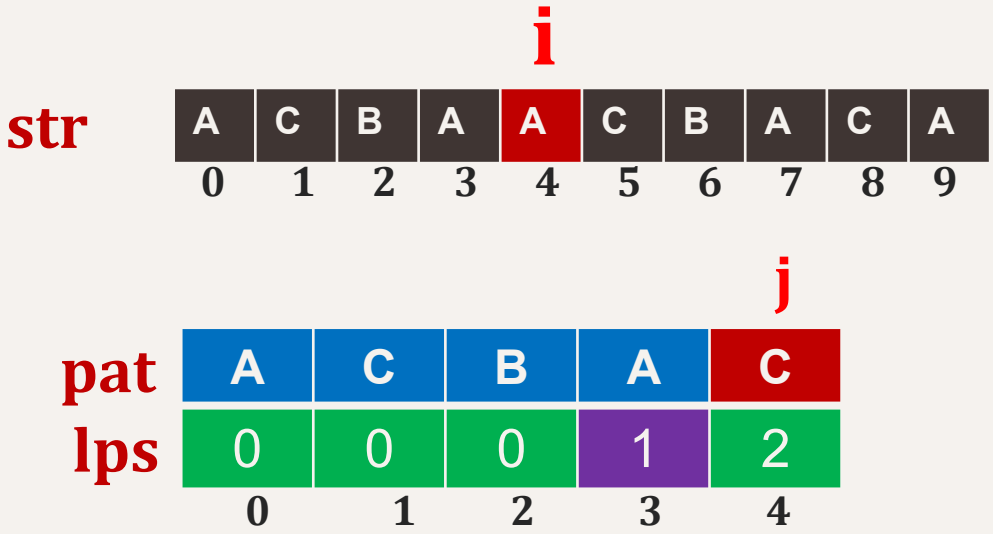
String: ACBAACBACA

Pattern: ACBAC



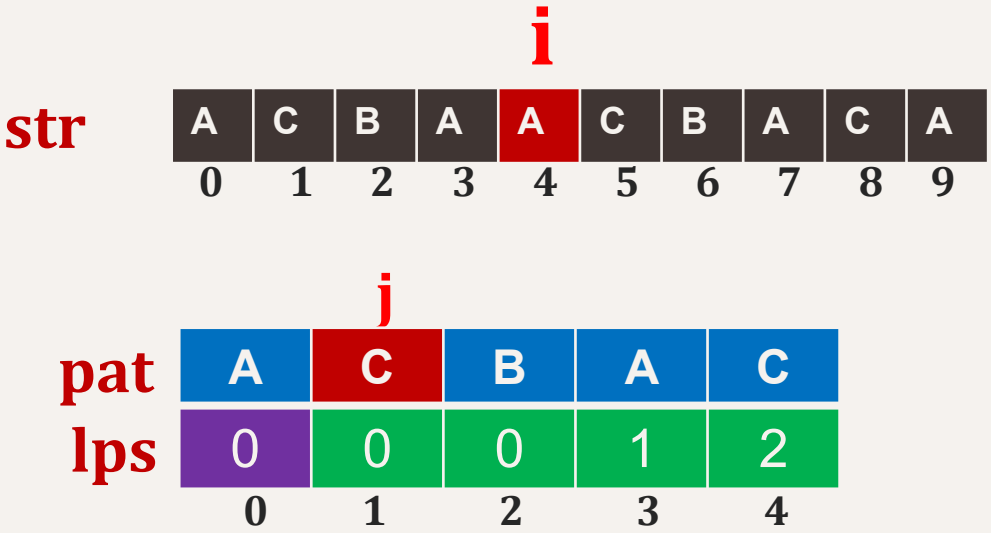
String: ACBAACBACA

Pattern: ACBAC



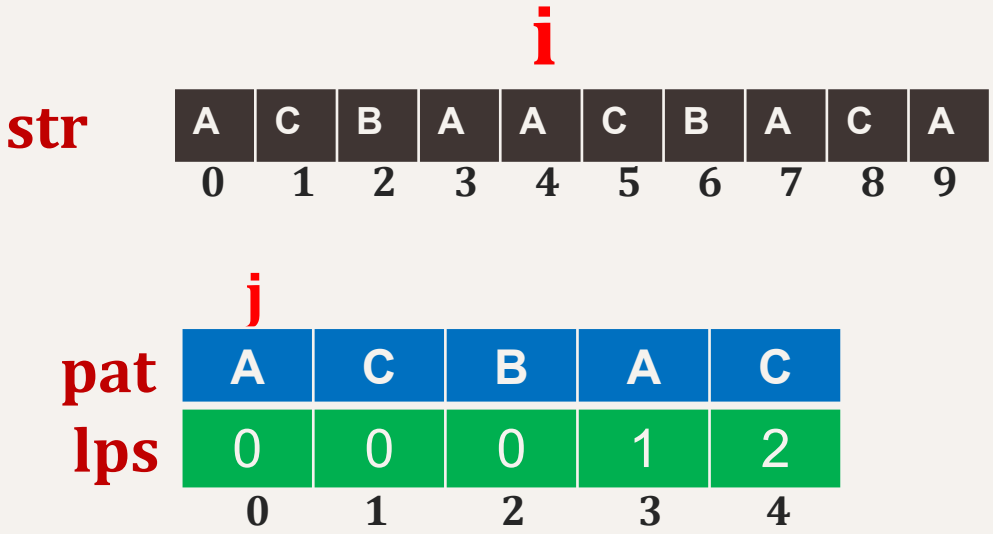
String: ACBAACBACA

Pattern: ACBAC



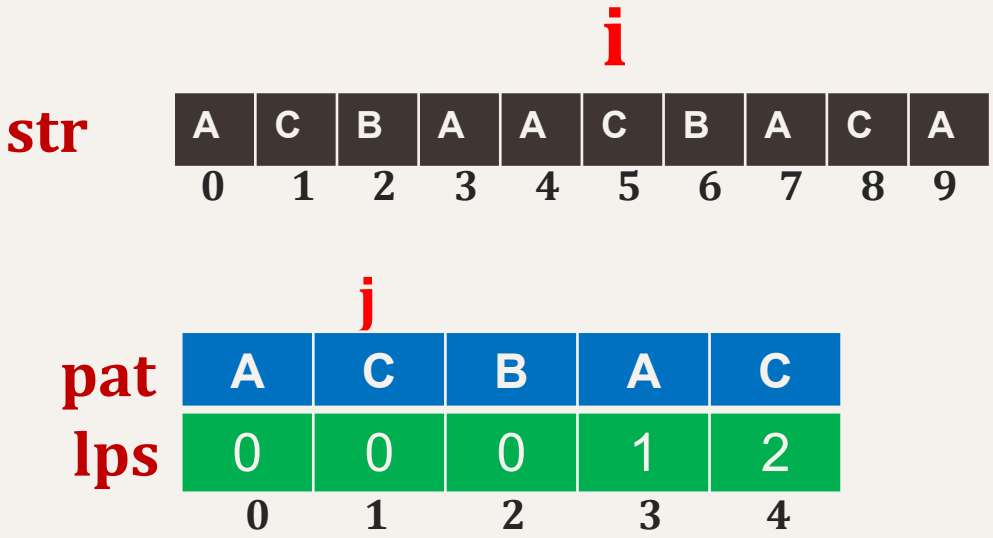
String: ACBAACBACA

Pattern: ACBAC



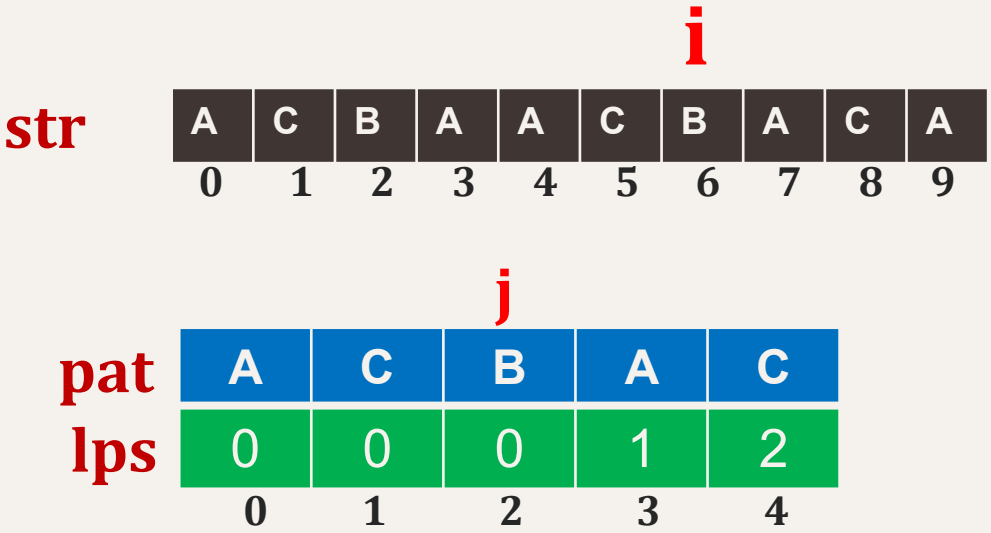
String: ACBAACBACA

Pattern: ACBAC



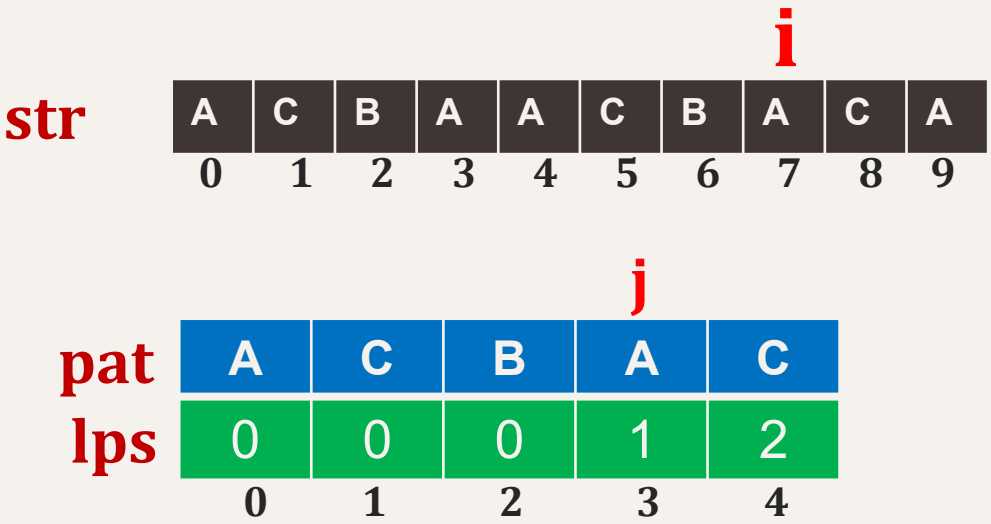
String: ACBAACBACA

Pattern: ACBAC



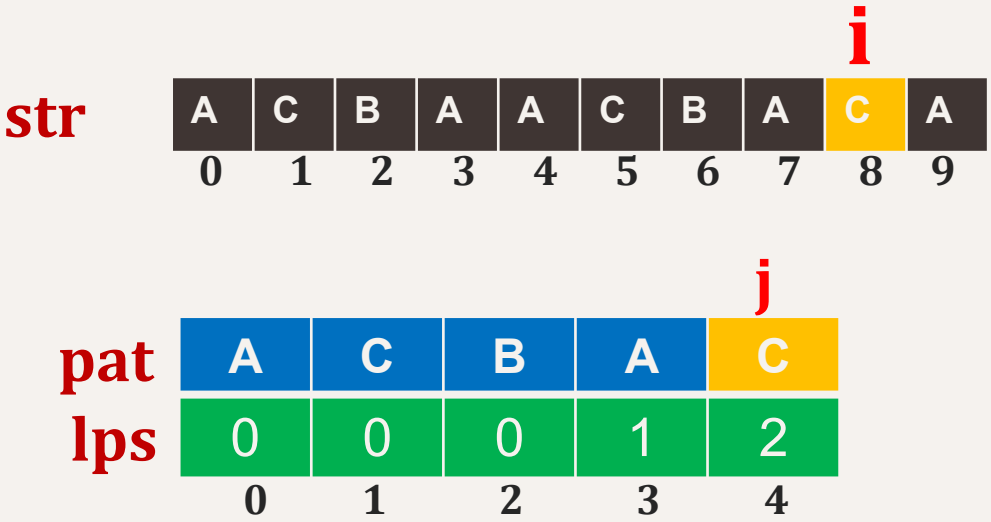
String: ACBAACBACA

Pattern: ACBAC



String: ACBAACBACA

Pattern: ACBAC



Time Complexity

```
vector<int> calc_ps(string pattern,int m) {

    vector<int> lps(p_len);
    int length = 0;
    int i = 1;
    while (i < m) {
        if (pattern[i] == pattern[length]){
            lps[i] = length + 1;
            length++; i++;
        }
        else {
            if (length != 0)
                length = lps[length - 1];
            else{
                lps[i] = length;
                i++;
            }
        }
    }
    return lps;
}
```

Running Time

O(1)

O(m)

Overall : O(m)

Time Complexity

```
bool kmp (string str, string pattern,int m,int n,int&
foundAt) {

    vector<int> lps = calc_ps(pattern,m);
    int i = 0, j = 0;

    while (i < n) {
        if (str[i] == pattern[j]){
            i++; j++;
        }
        else {
            if (j != 0)
                j = lps[j - 1];
            else
                i++;
        }

        if (j == m) {
            foundAt = i-m;
            j = lps[j-1];
            return true;
        }
    }
    return false;
}
```

Running Time

O(m)

O(n)

Overall : O(m+n)

Time Complexity

So, as can be seen from the previous runtime analysis, the runtime analysis of KMP's Algorithm is **O(m+n)** which is far better than the runtime of brute force algorithm **O(m*n)**