SNS COLLEGE OF ENGINEERING

Coimbatore-107

COURSE NAME: ANALYSIS OF ALGORITHM

II YEAR/ IV SEMESTER

UNIT – IV

STRING MATCHING ALGORITHM

Topic

Knuth Morris Pratt Algorithm

Unit 4 → Knuth Morris Pratt Algorithm

⇒ The basic idea is: to match the text with the pattern.

⇒ It is advantageous than Naiv string.

Disadvantages of Naive:

⇒ It checks each of pattern and text even if some part has already been matched.

⇒ If Mismatch occurs, it shifts the pattern by 1 and starts over.

⇒ Repeated Comparisons are done

Step 1:

⇒ Build Lps [Longest prefix suffix Array], before pereforming match.

⇒ When Mismatch occurs, kmp doesn't start over.

⇒ It uses Lps to skip charact & avoid repeating Comparisons.

Example:

Text = " A A A B A "
0 1 2 3 4

pattern = " A A B A "
0 1 2 3

Rules:

(1) If text[i] == pat[j] (match occurs)
⇒ increment i,j (i++ ; j++)

(2) If text[i] ≠ pat[j] (mismatch occurs)
⇒ Reset j = lps[j-1]
⇒ then increment i (i++).

Example:

	0	1	2	3	4	5	6	7	8	9	10
Text =	A	B	A	A	B	C	A	B	A	D	B

Pattern =	A	B	C	A	B
	0	1	2	3	4

Goal:

Find index of pattern in Text.

Step 1:

Find LPS Array (Largest Prefix Suffix Array)

Rules: Note (1).

1. All characters unique

eg: A B C D E

LPS Array = [0 0 0 0 0]

2. All characters same

eg: A A A A

LPS Array = [0 1 2 3]

Step 1:

LPS Array for pattern in Example

PATTERN = A B C A B (0 1 2 3 4 → Index)

(i). Split the given character as A, AB, ABC, ABCA, ABCAB

(ii) Calculate the prefix & Suffix for each term.

(iii) Update in LPS Array. (matched character/term length)

(iv) Always don't choose last matched longest Prefix & Suffix

LPS Array	Prefix & Suffix for Each term
1. A → 0 (Initial Always 0)	①. A B
2. B → 0	Prefix (Left to Right) : 1. A ; 2. AB X
3. C → 0	Suffix (Right to left) : 1. B X ; 2. AB X
4. A → 1	A ≠ B ; So B → 0
5. B → 2	

(No C! AB is matched in prefix & suffix of Term AB. But we should not choose last matched longest Prefix Suffix.) Update B → 0 in LPS Array for (mismatch)

②. Prefix & Suffix of "ABC":

Prefix	← Compare →	Suffix
1. A	≠	C X
2. AB	≠	BC X
3. ~~ABC~~	Should not Consider	~~ABC~~

∵ A ≠ C ; Set C → 0 in LPS Array (because of mismatch)

③. Prefix & Suffix of "ABCA" :

Prefix	← Compare →	Suffix	
1. A	==	A	✓ match
2. AB	≠	CA	X
3. ABC	≠	BCA	X
4. ~~ABCA~~	Don't Consider	~~ABCA~~	

∵ A == A (matches) put A → 1. in LPS Array (length of char)

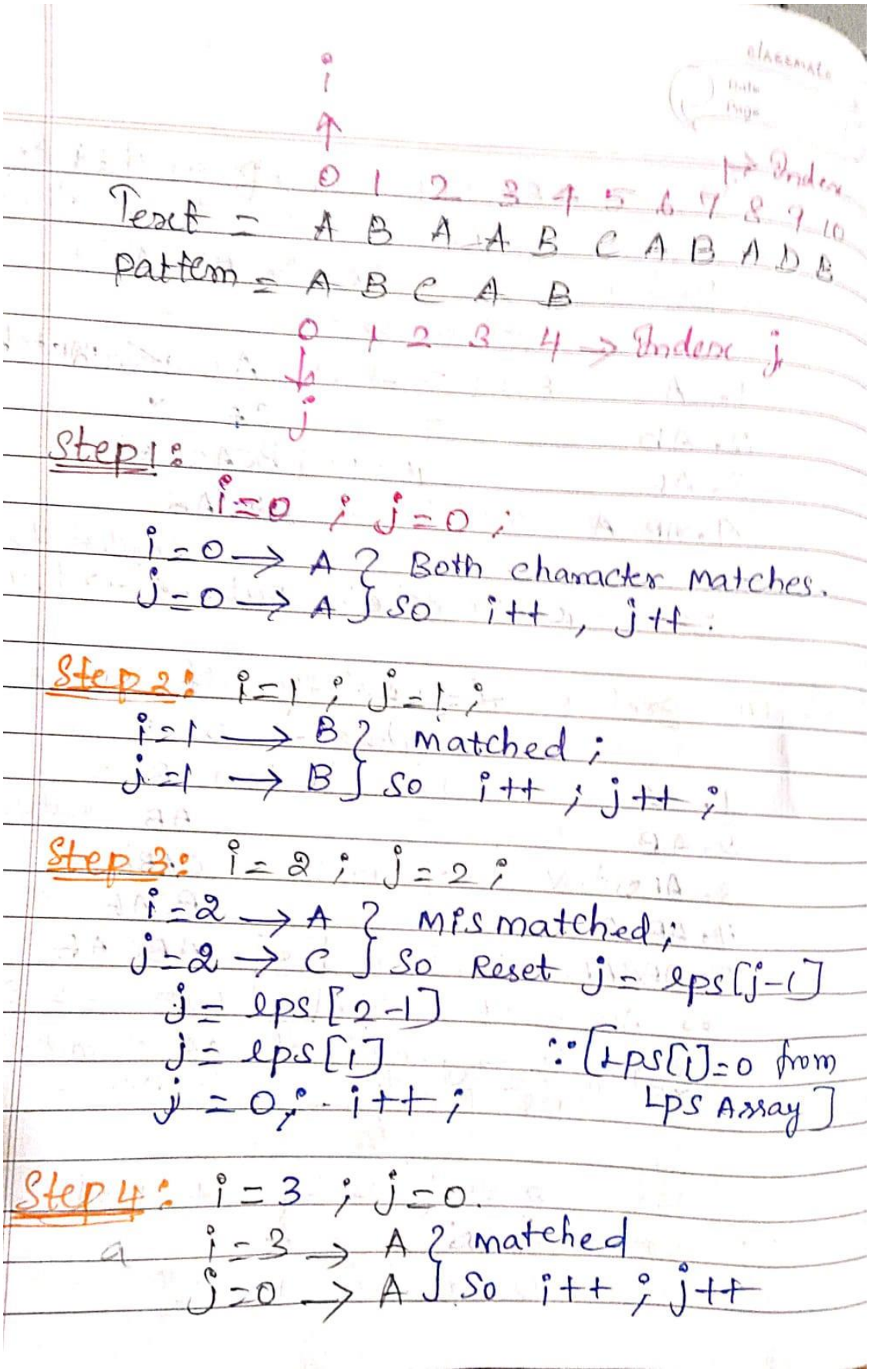
④. Prefix & Suffix of ABCAB :

Prefix	← Compare →	Suffix	
1. A	≠	B	X
2. AB	==	AB	(Match) ✓
3. ABC	≠	CAB	X
4. ABCA	≠	BCAB	X
5. ~~ABCAB~~	Don't Consider	~~ABCAB~~	

∵ AB == AB (matches) ; put B → 2 in LPS Array. (i.e. length of AB (character))

LPS Array

Index →	0	1	2	3	4
Pattern →	A	B	C	A	B
LPS Array →	0	0	0	1	2


i
0 1 2 3 4 5 6 7 8 9 10 → Index
Text = A B A A B C A B A D B
pattern = A B C A B
0 1 2 3 4 → Index j
j
Step 1:
i=0 ; j=0 ;
i=0 → A } Both character matches.
j=0 → A } So i++, j++.
Step 2: i=1 ; j=1 ;
i=1 → B } matched ;
j=1 → B } So i++ ; j++ ;
Step 3: i=2 ; j=2 ;
i=2 → A } Mismatched ;
j=2 → C } So Reset j = lps[j-1]
j = lps[2-1]
j = lps[1] ∵ [LPS[1]=0 from LPS Assay]
j=0 ; i++ ;
Step 4: i=3 ; j=0.
i=3 → A } matched
j=0 → A } So i++ ; j++

Step 5: i = 4 ; j = 1

i = 4 → B ? matched ;
j = 1 → B } So i++ ; j++ ;

Step 6: i = 5 ; j = 2

i = 5 → C ? Matched ;
j = 2 → C } So i++ ; j++ .

Step 7: i = 6 ; j = 3

i = 6 → A ? matched
j = 3 → A } So i++ ; j++

Step 8: i = 7 ; j = 4

i = 7 → B ? Matched
j = 4 → B } So i++ ; j++

Now j = 4 reached max length of pattern.

If (j = Length of pattern)
We found pattern in text at index 3.

Note (X-)

Incrementi; i++; But Reset j = 0.

(This is done because text length is not completed) so Search is continued.

Step 9: j=0 ; i = 8

i=8 → A } matched
j=0 → A } so i++ , j++.

Step 10: j=1 ; i=9.

i=9 → D } mismatched
j=0 → B }

Reset j = lps[i-1] [∵ lps[0]=0]
j = lps[1-1] = lps[0]
∴ j=0 ; i++

Step 11: j=0 ; i=10.

→ B
Since i=10; End of text.
j=0 → A & mismatch.

← Stop here →

Now we found pattern "ABCAB" is matched with text at index 3 & 6 in only position.

```
Algorithm LpsArr(pattern, m, lps[])
{
while (i<m)
if(pattern[i]==pattern[len])
{ len++;
  lps[i] = len;
   i++;
} else {
if(len!=0)
{ len = lps[len-1];
} else
 { lps[i] = 0;
   i++
 }}}}
Algorithm kmp(pattern, text)
{
m = length of pattern;
N = length of text;
while (i<N)
{
if (pattern[i]==text[i])
{
i++;
j++;
}
```

```
if (j==m)
{
// pattern found at index [i-j];
j = lps[j-1];
} else if (i<N && pattern[j] != text[i])
{
if (j!=0)
j = lps[j-1];
else
i++;
}}}
```

Time Complexity:

1) lps Array Construction = O(m)
2) Pattern Search = O(n)
Total = O(n+m)

* n → text length * m → pattern length

Efficient:

* No Backtracking in text *. Avoids Rechecking characters * lps tells where to resume in pattern during mismatch

* Space Complexity:

*. Much faster than brute force O(n*m) in worst case
*. O(m) for lps Array