UNIT-6

String matching

Naive string matching

* String matching means finding all occurrences of a pattern in a given text [either number series or alphabets]
* Naive pattern searching is the simplest method among other pattern searching algorithm. it checks for all character of the main string to the pattern. it finding one or all exact occurrences of a pattern in a text.
* this Algorithm is helpful for smaller texts. we can find substring by checking once for the string.
* the naive approach tests all the possible placement of pattern $P[1.....m]$ relative to text $T[1.....n]$. we try to shift $s = 0, 1, 2, \ldots, n-m$, successively and for each shift s.

  Compare $T[s+1, \ldots s+m]$ to $P[1.....m]$.

```
1. n ← length[T]
2. m ← length[P]
3. for s ← 0 to n-m
4. do if P[1....m] = T[s+1, .... s+m]
5. then print "pattern occurs with shift"
```

Analysis:- for loop executes $n-m+1$ times ∴ Time complexity = $O(n-m+1)$

Q1 Text T = a c a a b c

pattern p = a a b , the pattern is found at which index using naive string matching Algorithm.

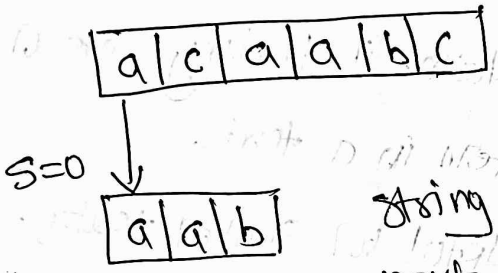
Sol Text T =

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

pattern p =

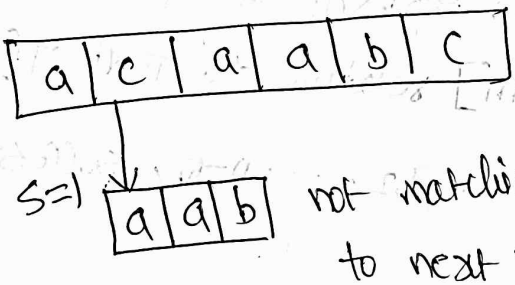
a	a	b
0	1	2

Step1 :- matching starting from 0 index, initially shift s=0



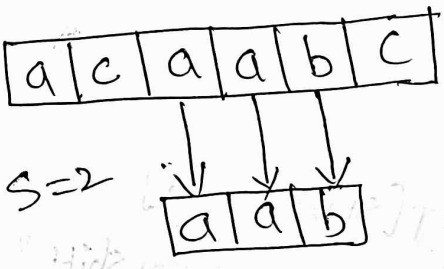
string not matching, then shifting to next index.

Step2:- pattern is shifting to next index. s=1



not matching, then pattern is shifting to next index.

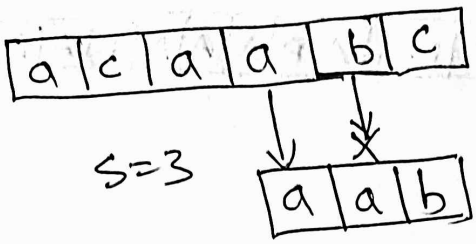
Step3:- pattern is shifting to next index s=2



1st in the given pattern 1,2,3 characters are matching with the given text

∴ pattern is found at index 2

step4:- Again pattern is shifting to next index



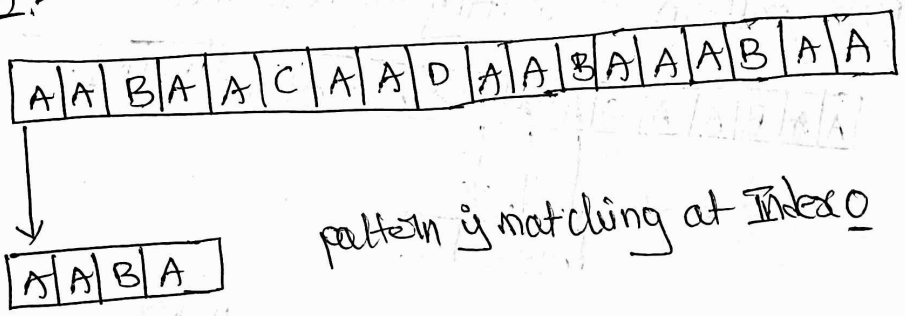
In a given pattern 1st character is matching, 2nd character is not matching with given Text

∴ pattern aab is found at index 2

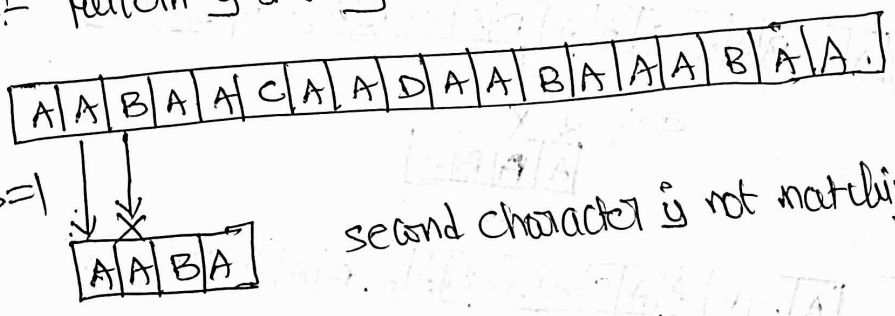
Q12:- Text T = AABAACAADAABAAABAA

pattern P = AABA

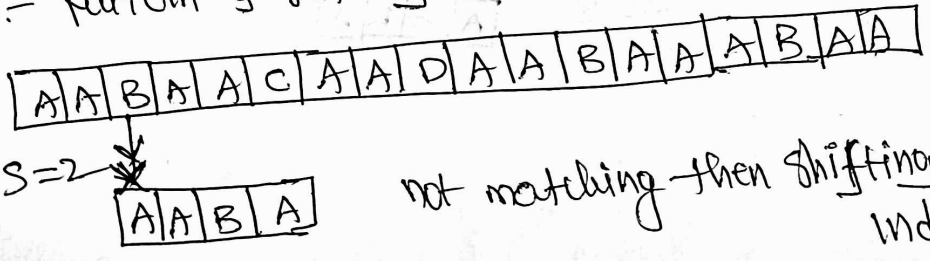
Sol:- step1:-



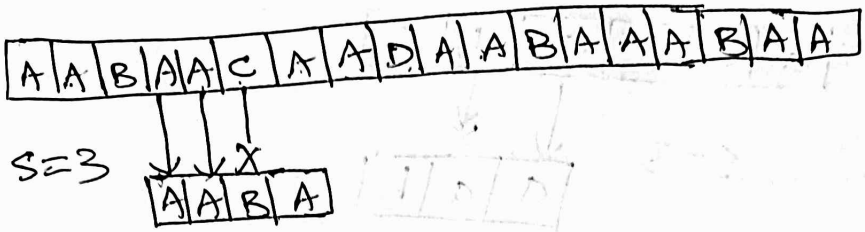
step2:- pattern is shifting to next index



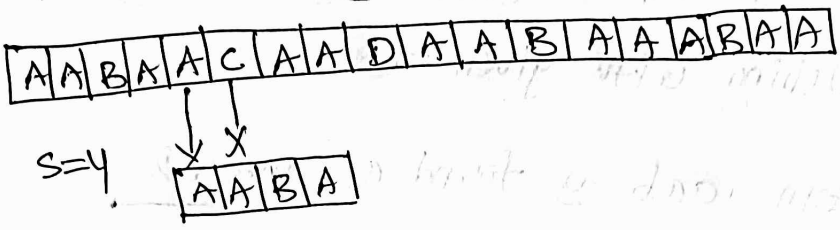
step3:- pattern is shifting to next index



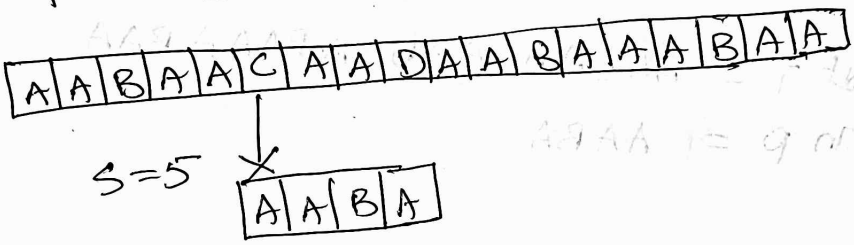
step4:- pattern is shifting to next index.



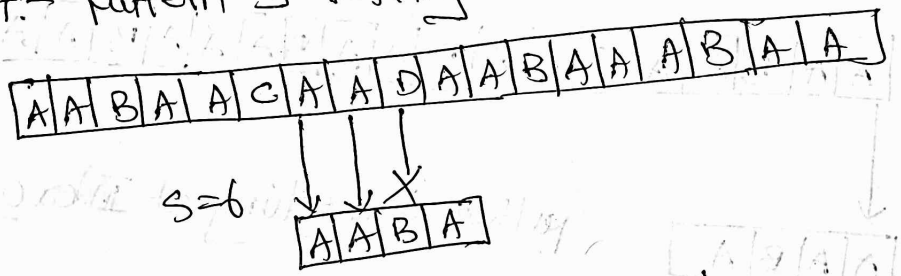
step5:- pattern is shifting to next index



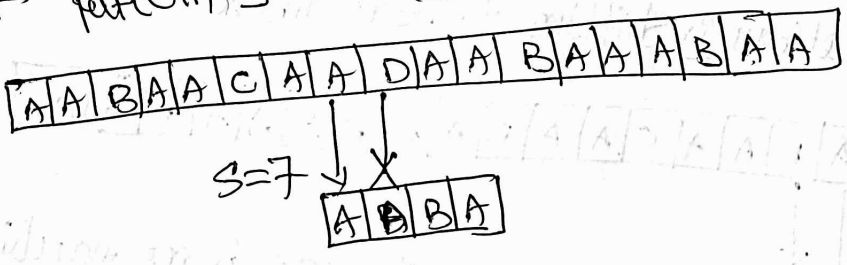
step6:- pattern is shifting to next index



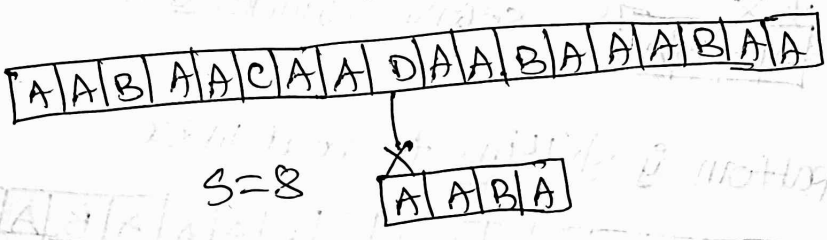
step7:- pattern is shifting to next index



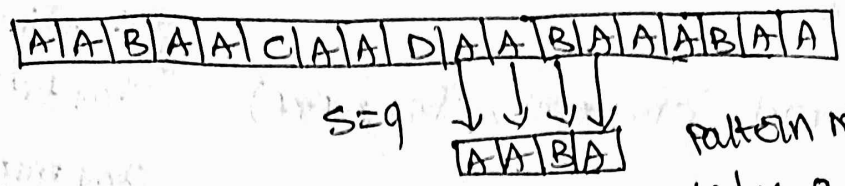
step8:- pattern is shifting to next index



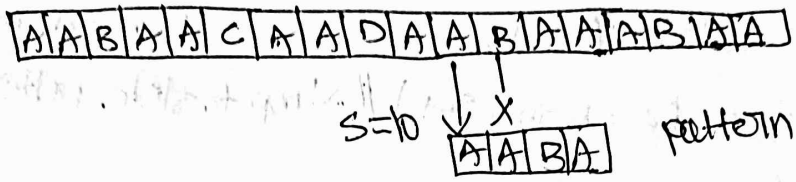
step9:-



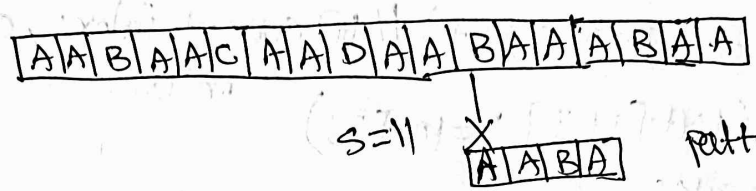
step10:-



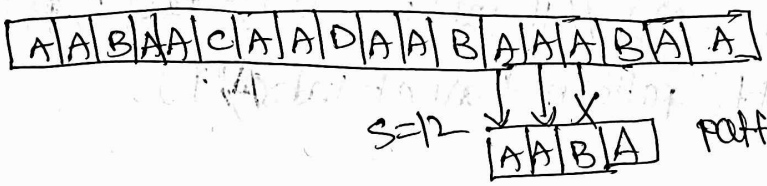
step11:-



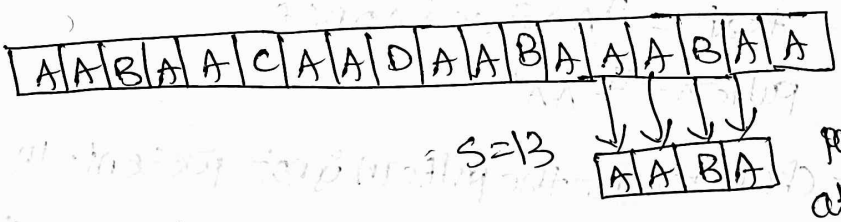
step12:-



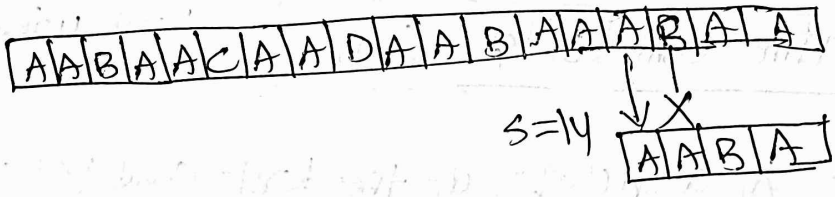
step13:-



step14:-



step15:-



∴ pattern matching at index 0

" " " index 9

" " " index 13

Algorithm

```
void search (char *pat, char *txt)
{
  int M = strlen (pat);
  int N = strlen (txt);
  for (int i=0; i<=N-M; i++) // A loop to slide pattern one by one
  {int j;
    for(j=0; j<M; j++) // for current index i, check for pattern match.
      if (txt[i+j] != pat[j])
       break;
    if(j==M) // if pat[0.....M-1] = txt[i,i+1 .. i+M-1]
     printf(" pattern found at index %d", i);
  } }
```

```
int main()
{char txt[] = "AABAACAADAA
               BAAABAA"
 char pat[] = "AABA"
 search (pat, txt);
 return 0;
}
```

Best case:- text = "AABCCAADDEE"
pattern = "FAA"

the first character of the pattern is not present in text at all. the pattern is just move the window, the time complexity is the text window size.

∴ Time Complexity = O(n).

Worst case:- All characters of the text and pattern are same

text = "AAAAAAAAAA"
pattern = "AAAA"

in each window we need to compare all the characters until the last window.

∴ Time Complexity = O(m × (n−m+1))
(m → window size; (n−m+1) → no. of windows)

If the last character is different even though we need to check all character in the window. Ex: text = "AAAAAAAAAAAB"
pattern = "AAAB"

* Note:- pattern searching is an important problem in computer science. when we do search for a string in notepad/word file browser (or) database (or) google search, pattern searching Algorithms are used to show the search Results

Tries string matching

* Trie is a Tree that stores a set of strings. how we are going to store a string in a tree. A Trie is a tree like information Retrieval data structure whose nodes store the letters of an Alphabet.
* every node except root node will store a letter in the Alphabet. it is known as Radix Tree (or) Digital Tree (or) prefix Tree

Ex $S = \{ bear, bell, bed, bull, stop, stock \}$

* Trie is an efficient information Retrieval data structure it is also called Digital Tree and sometimes called Radix Tree (or) prefix Tree.

* Trie is pronounced as "try" it comes from the word "Retrieval".

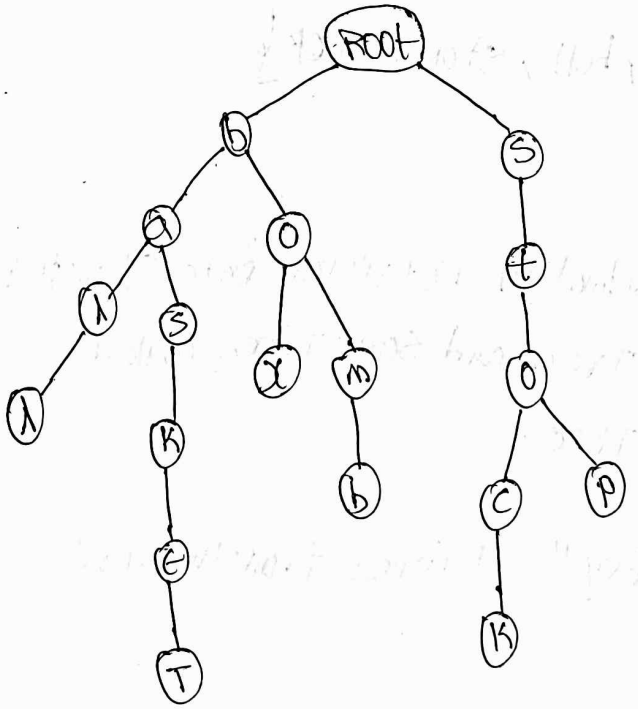
* there are 3 types of tries

1. standard Trie
2. compressed Trie
3. suffix Trie

Standard Trie:-

* it is an ordered Tree like Data structure.
* Each node [except the Root node] in a standard trie is labeled with a character
* the children of a node are in Alphabetical order
* Each node of branch Represents a possible character of keys of words.
* Each node of branch may have multiple ~~characters~~ branches
* the last node of every key of word is used to mark the end of word of node.

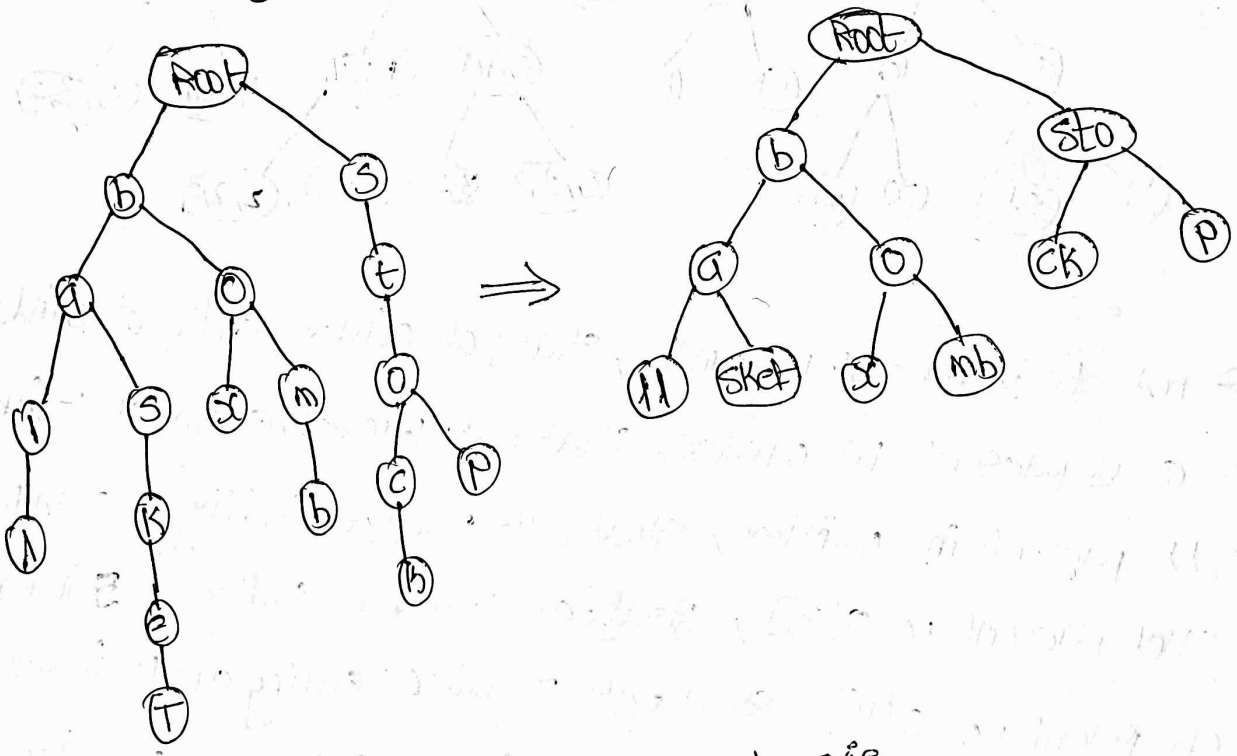
Ex. String S = {ball, box, bomb, basket, stock, stop}



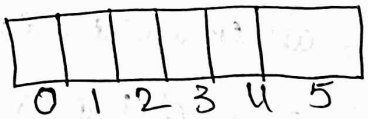
Compressed Trie

* it's a advanced version of standard Tries. If we have a node with one child, we try to merge them to get the easy form of the Trie.
* Each node [Except the leaf nodes] have atleast 2 children. it's used to Achieve space optimization.

Ex String S={ ball, box, bomb, basket, stock, stop }



* Representation of a Compressed Trie



s[0] = b a l l
s[1] = b o x
s[2] = b o m b
s[3] = b a s k e t
s[4] = s t o c k
s[5] = s t o p

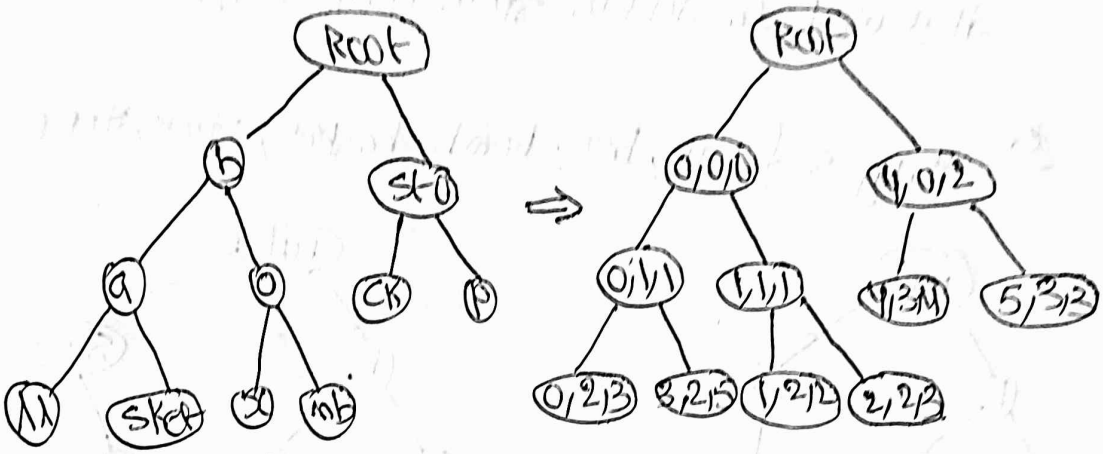
* we Represent a Compressed Trie in three tuple notation

node(i, j, k)

→ i Represents index at S at which the word is present

→ j Represents the index at which the prefix starts

→ k represents the index at which the prefix ends.



→ node b is present in 0 index, starts at 0 index, ending at 0 index

→ a is present in 0 index, starts at 1-index, ending at 1-index

→ 11 present in 0-index, starts at 2-index, ending at 3-index

→ sket present in S[3], starts at 2-index, ending at 5-index.

→ ck present in S[4], starts at 3-index, ending at 4-index

→ p present in S[5], starts at 3-index, ending at 3-index.

→ st0 present in S[4], prefix starts at 0-index, ending at 2-index.

→ 0 present in S[1], prefix starts at 1-index, ending at 1-index

→ x present in S[1], prefix starts at 2-index, ending at 2-index

→ mb present in S[2], prefix starts at 2-index, ending at 3-index

Suffix Trie

* Suffix Trie is a compressed trie containing all the suffixes of the given Text-[string]. Suffix Trie helps in solving a lot of string Related problems like pattern matching, finding distinct substrings in a given string. finding the longest palindrome
* we have to build the suffix Trie for the string "minimize"

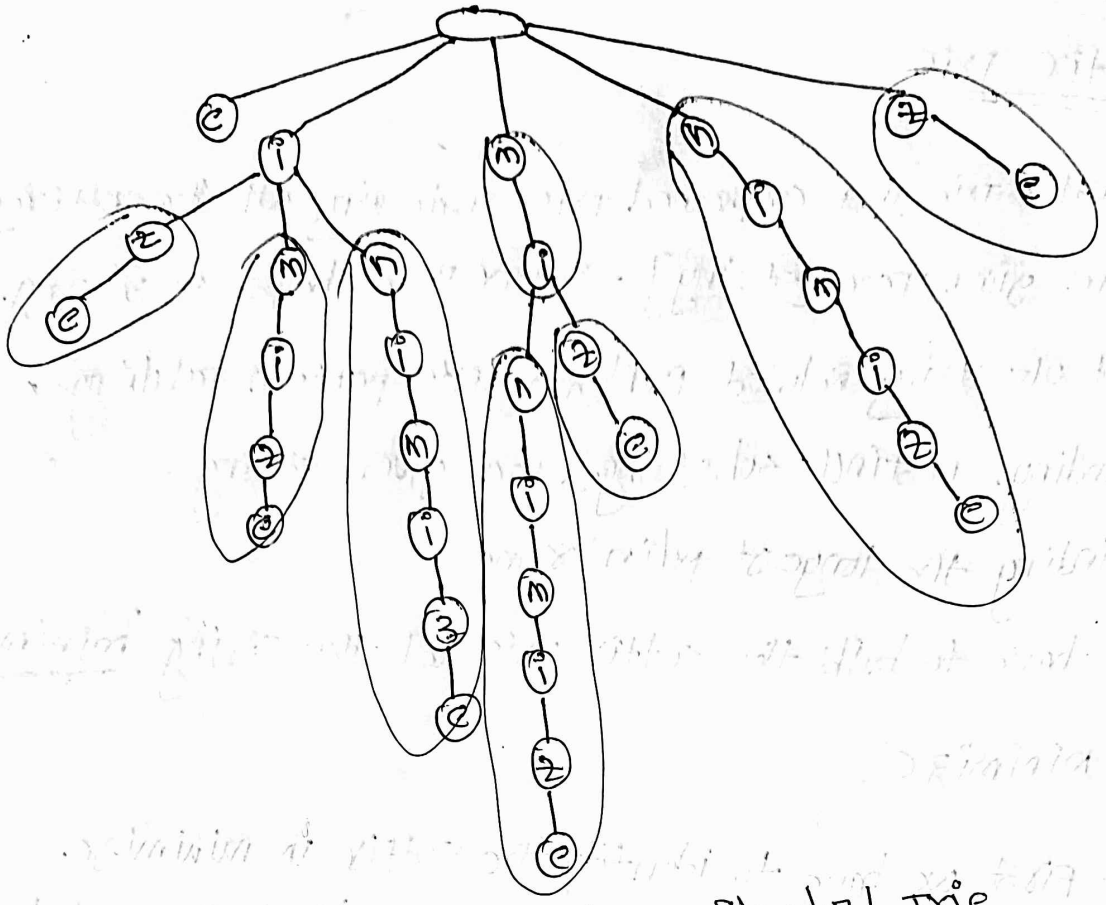
Ex minimize

Sol:- First we have to identify the suffix in minimize. suffix is "e", then include "z" becomes "ze", then include "i" becomes "ize".

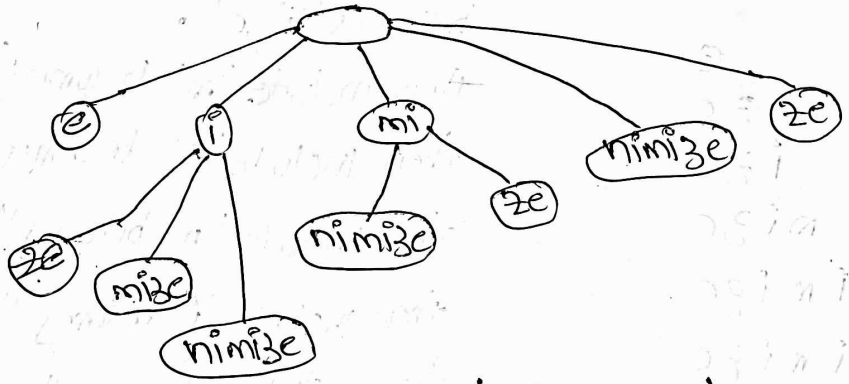
e
ze
ize
mize
imize
nimize
inimize
minimize

then include 'm' becomes "mize"
then include 'i' becomes "imize"
then include 'n' becomes "nimize"
then include 'i' becomes "inimize"
then include 'm' becomes "minimize"

Then we can construct the standard Trie for this string

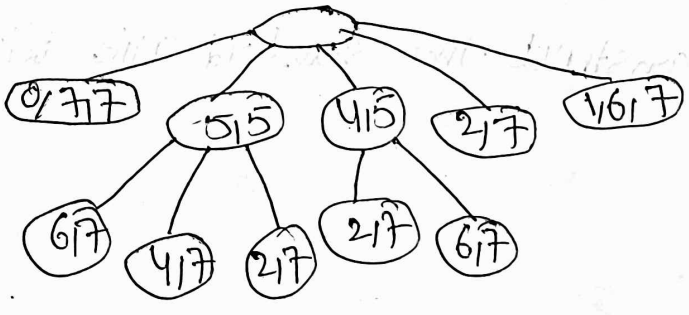


Compressed Trie of the above Standard Trie



the compressed Trie can be represend in the Tuple Representation

0	1	2	3	4	5	6	7
m	i	n	i	m	i	z	e



s[0] = e
s[1] = ze
s[2] = ize
s[3] = mize
s[4] = imize
s[5] = nimize
s[6] = inimize
s[7] = minimize

Rabin–Karp Algorithm [Rabin & Karp in 1987]

* Rabin-Karp Algorithm is used for searching (or) matching patterns in the Text [string] using a hash function.
* the hash function is a tool to map a larger input value to a smaller output value. This output is called the hash value.
* A sequence of characters is taken and checked for the possibility of the presence of the required string.
* If the possibility is found then character matching is performed.
* pattern hash value and Text window hash value both are matching then character matching is performed. otherwise no character matching is performed.

Ex. Text = A B C C D D A E F G and
pattern = C C D

Sol:- n is the length of the pattern and
m is the length of the Text

n=3 and m=10

d is the number of characters in the input set Text
d = 10.

Text window: [A B C] C D D A E F G
pattern C C D

* Calculate the hash value of the pattern "CCD"

hash value for pattern (P) = $\Sigma (v * d^{m-1}) \cdot \text{mod } 13$

A – 1
B – 2
C – 3
D – 4
E – 5
F – 6
G – 7
H – 8
I – 9
J – 10

∴ v ⇒ the ASCII value of the character and choose any prime number [here we use 13] in such a way that we can perform all the calculations

$$\text{pattern (P)} = ((3 \times 10^{3-1} + 4 \times 10^{3-2} + 4 \times 10^{3-3}) \text{ mod } 13)$$

$$= (3 \times 10^2 + 4 \times 10^1 + 4 \times 10^0) \text{ mod } 13$$

$$= (300 + 40 + 4) \text{ mod } 13$$

$$\text{pattern (P)} = 344 \text{ mod } 13 = \underline{1}$$

* Calculate the hash value of the text window of size m.

for the first Text window ABC

$$\text{hash value for Text (T)} = (1 \times 10^{3-1} + 2 \times 10^{3-2} + 3 \times 10^{3-3}) \text{ mod } 13$$

$$= (1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0) \text{ mod } 13$$

$$= (100 + 20 + 3) \text{ mod } 13$$

$$\text{Text (T)} = 123 \text{ mod } 13 = \underline{6}$$

* Compare the hash value of the pattern with the hash value of the Text. If both are matching then only we can check the character-matching. otherwise no need to check character-matching.

* In the example, the hash value of the First window and hash value of the pattern both are matching then we can check the character-matching

Text First window = A B C D B A B F G

pattern = C D D

not matching then go to the next Text window.

Step2:- If not matching then Text index is incremented (i++)
. pattern index Remains same

Text second window = A B C D B A B F G

pattern = C D D

* hash value of the next window Text is

$$\text{pattern}(T) = (2\times10^{3-1} + 3\times10^{3-2} + 4\times10^{3-3}) \bmod 13$$
$$= (2\times10^{2} + 3\times10^{1} + 4\times10^{0}) \bmod 13$$
$$= (200 + 30 + 4) \bmod 13$$
$$\text{pattern}(T) = (234) \bmod 13 = \underline{12}$$
$$= 233 \bmod 13 = \underline{12}$$

* hash value of the pattern and the hash value of the Text second window [6 ≠ 12] both are not matching.

Then Text widow is changing. window is moving to next window.

Step3:- Text window = A B C C D DA C F G

pattern = C D D

* hash value of the Text window is =

$= (3 \times 10^{3-1} + 9 \times 10^{3-2} + 4 \times 10^{3-3}) \bmod 13$

$= (3 \times 10^{2} + 9 \times 10^{1} + 4 \times 10^{0}) \bmod 13$

$= (300 + 90 + 4) \bmod 13$

$= (394 - 2) \bmod 13$

$= 332 \bmod 13 = 7$

* hash value of the Text window and the value of the pattern both are not matching (6≠7)

∴ then window is moving one step [i value is incremented]

Step4:- Text next window = A B C C D D A C F G

pattern = C D D

* hash value of the next Text window is

hash value of pattern (p) $= (3 \times 10^{3-1} + 4 \times 10^{3-2} + 4 \times 10^{3-3}) \bmod 13$

$= (3 \times 10^{2} + 4 \times 10^{1} + 4 \times 10^{0}) \bmod 13$

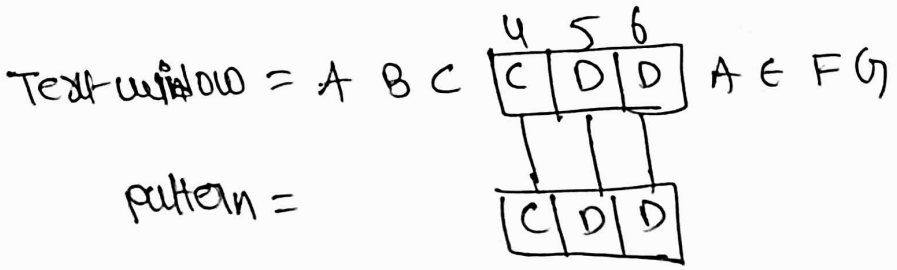
$= (300 + 40 + 4) \bmod 13$

$= (344 - 3) \bmod 13$

hash (p) $= 331 \bmod 13$

$= 6$

∴ hash value of the next Text window and the hash value of the pattern ($6=6$) both are same. then we can check the character matching.



∴ matching found at 4th index.

Continue the procedure for the next characters in the Text.

Steps:-

Knouth morris pratt [kmp] Algorithm

* Kmp Algorithm is used to find a "pattern" in a "text". this Algorithm compares character by character from left to Right.
* But whenever a mismatch occurs, it uses a preprocessed table called "prefix Table" to skip characters comparison while matching.
* the Basic pattern matching Algorithm doesn't study the pattern ~~doesn't observe the pattern~~ what is there in the pattern as it is blindly it is Checking every Alphabet, just shifting this to the Right incase of mismatch.
* the Basic idea of Kmp Algorithm is to, just study the pattern and Reduce some time, we can see the idea of kmp algorithm.

Ex pattern: a b c d a b c
1 2 3 4 5 6 7

prefix a, ab, abc, abcd ..

suffix c, bc, abc, dabc ..

the idea of this kmp is to inside the string is there prefix same as suffix.

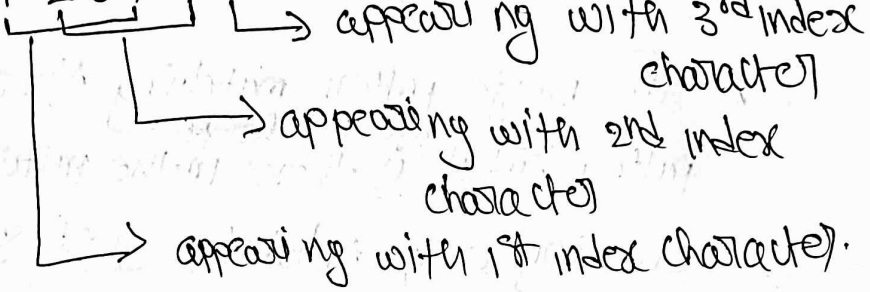
pattern: abcd abc
same

* the idea is that the beginning of the pattern is appearing again anywhere else in the pattern or not. that is KMP algorithm. prefix Table. now see how to prepare prefix Table.

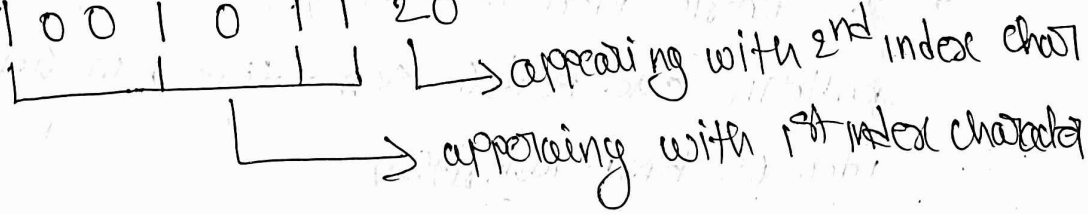
Eg: P1: a b c d a b e a b f
0 0 0 0 1 2 0 1 2 0

a is 1st index, a is appearing in two times
b is 2nd index, b is appearing in two times.

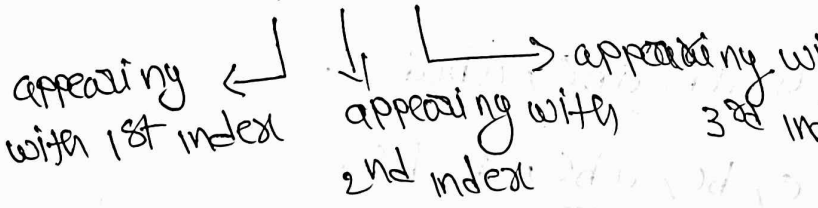
Eg: a b c d e a b f a b c
0 0 0 0 0 1 2 0 1 2 3



Eg: a a b c a d a a b e
0 1 0 0 1 0 1 1 2 0



Eg: a a a a b a a c d
1 2 3 1 2

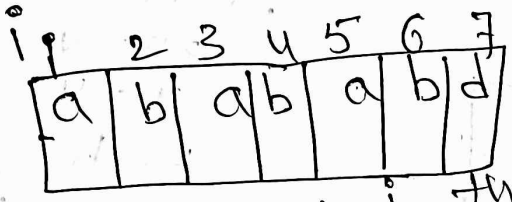


Ex:- Text: a b a b a b d

pattern: a b a b d

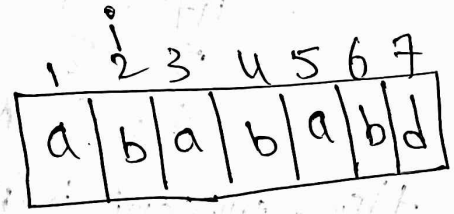
Sol:- First prepare the prefix Table (or) Pi (π) Table for the given pattern.

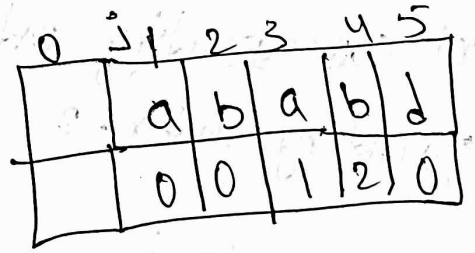
j	0	1	2	3	4	5
		a	b	a	b	d
		0	0	1	2	0



→ Now compare j+1=i then j, i increment and i also incremented.

Step1:- j+1=i ⇒ 0+1=1 ⇒ a=a then j, i values increment



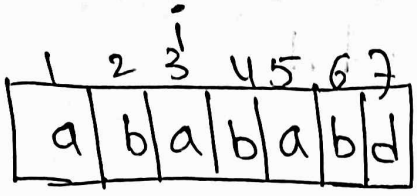


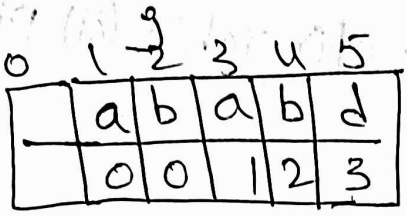
∴ j+1=i

b=b matching

then i, j values are incremented

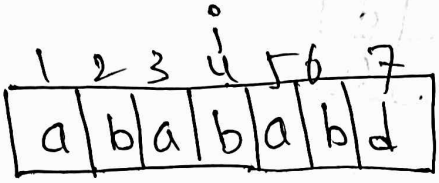
step2:-

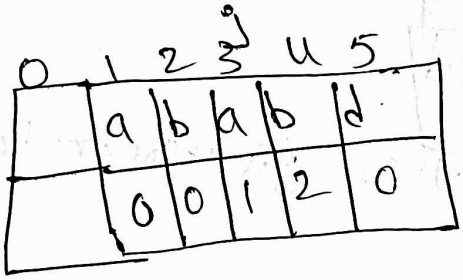




$j+1 = i$
$2+1 = 3$
$a = a$
matching then i, j values are incremented.

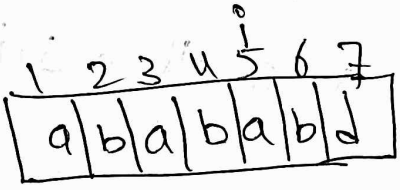
step3

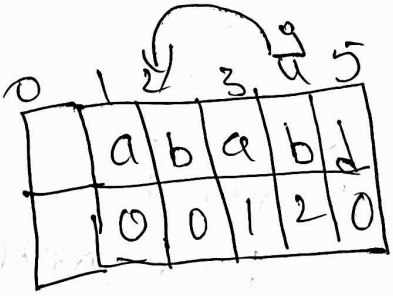




$\therefore j+1 = i$
$3+1 = 4$
$4 = 4$
$b = b$
matching then i, j values are incremented.

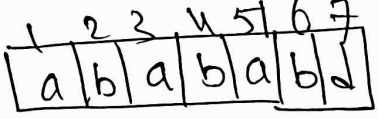
step4

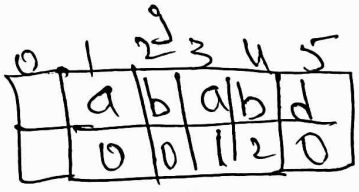




$\therefore j+1 = i$
$4+1 = 5$
$5 = 5$
$d = a$ not matching
then present j in 4th index.
that corresponding index is 2
then j is moving to corresponding index: 2

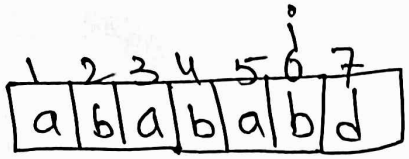
step5

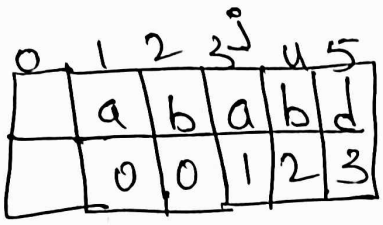




$\therefore j+1 = i$
$2+1 = 5$
$3 = 5$
$a = a$ matching
then i, j values are incremented

Step6:-





$\therefore j+1 = i$

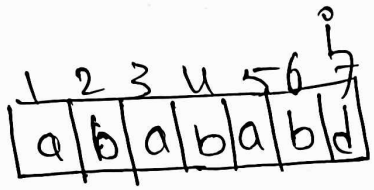
$3+1 = 6$

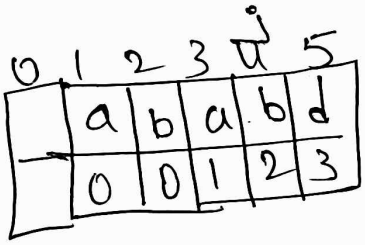
$4 = 6$

$b = b$

matching then i, j values are incremented.

Step7:-





$\therefore j+1 = i$

$4+1 = 7$

$5 = 7$

$d = d$

matching then i, j values are incremented.

Q) Text : a b a b c a b c a b a b a b d

pattern : a b a b d.

Sol) First prepare the prefix Table.

0	1	2	3	4	5
	a	b	a	b	d
	0	0	1	2	0

we can add prefix 0th index for performing operation

Boyer moore Algorithm

* Boyer moore Algorithm is the Fastest way to Find the pattern (or) matching the pattern in a given string (or) Text.
* this Algorithm is Finding the pattern in a string. it compares each character of pattern to find a word of the same characters into the string.
* when character don't match, the search Jumps to the next matching position in the pattern by the value indicated in the **Bad match Table**
* the Bad match Table indicates how many Jumps should it move from current position to the next.
* how to create the Bad match Table, First step is calculate the value of each letter of the pattern to create the Bad match Table using the Formula

$$\text{value} = [\text{length of pattern} - \text{index of each letter in the pattern} - 1]$$

Ex:

0	1	2	3
a	b	c	d

value (a) = [4 − 0 − 1] = 3
value (b) = [4 − 1 − 1] = 2
value (c) = [4 − 2 − 1] = 1
value (d) = [4 − 3 − 1] = 0

Letter	a	b	c	d	*
value	3	2	1	0	4

value of last letter and other letters that not in substring will be the length of pattern [4]

* After that, you can compare the substring [pattern] and the string [Text].
* you start from the index of the end letter in the pattern. In this case the letter is "J"
* If the letter matches, then compare with the preceeding letter "C" in this example.
* it don't match, check it's value in the Bad match Table.
* then skip the number of positions that the table value indicates.
* Repeat the steps until all the letters match.

Ex: Text T = This is a TEST
pattern P = TEST

Sol: (skip) First construct Bad match Table using the pattern.

Bad match Table

letter	T	E	S	*
value	3	2	1	4

* the pattern Contains Repeated character 'T'.
* Bad match table don't Contains Repeated characters values can be calculated by using the equation

value = [length of pattern − index of each letter in the pattern − 1]

value (T) = [4 − 0 − 1] = 3

value (E) = [4 − 1 − 1] = 2

value (S) = [4 − 2 − 1] = 1

Step2:-

Text T = | T | H | I | S | | I | S | | A | | T | E | S | T |

Pattern P = | T | E | S | T | not matching

But 'S' is in Bad match Table with the value '1'. means pattern is moving one position to the Right

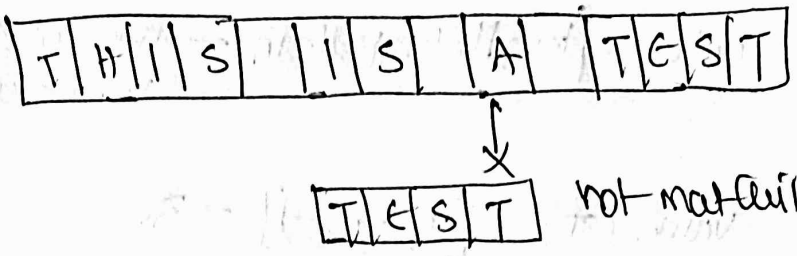
Step3:- Text = | T | H | I | S | | I | S | | A | | T | E | S | T |

| T | E | S | T | not matching.

* If it is empty character, then ~~search~~ consider Last character '*' in a Bad match Table. Empty character is not there in a Bad match Table. then consider Last character '*'. That indicate value 4.

* then pattern is moving 4 positions to the Right.

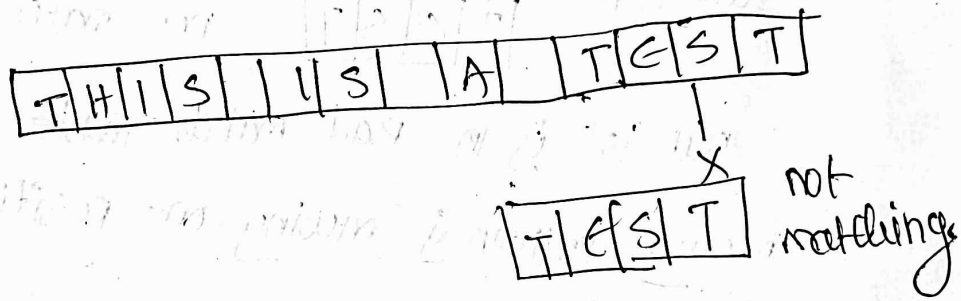
Step1: Text:



not matching.

and character A is also not there in Bad match Table. Then Consider last character '*' in a Bad match Table, the value is 4. then pattern is moving 4 positions to the Right

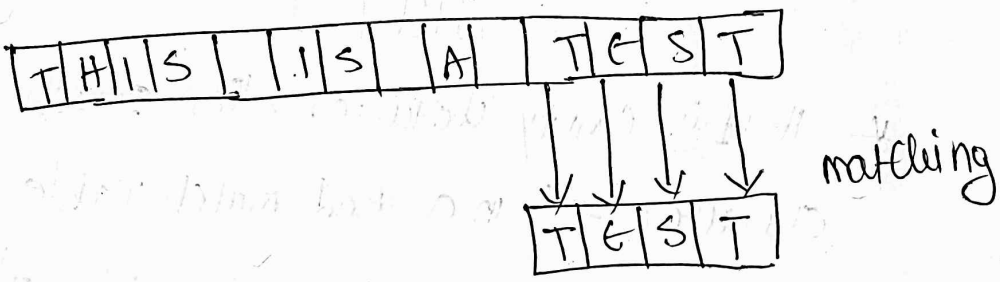
Steps:



not matching

But is in Bad match Table with the value '1'. means pattern moving one step to Right

Step3:



matching

then Compare the prefixes one by one.

∴ patter is matching at 10th position in a given Text.