UNIT- V
Pattern Matching Algorithm

A string is a sequence of characters. Let

Text[0,…n-1] be the string of length n, and

Pattern[0,…,m-1] be some substring of length m, then

Pattern matching is a technique of finding the substring in text which is equal to pattern is called Pattern matching.

Applications of Pattern Matching Algorithm

1 : Pattern matching technique is used in text editors.

2 : Search engines use the pattern matching algorithm for matching the query submitted by the user.

3 : In biological research pattern matching algorithm is used.

Different types of Pattern Matching Algorithm

1. Brute Force Algorithm
2. Boyer Moore Algorithm
3. Knuth-Morris-Pratt Algorithm

1. Brute Force Pattern Matching Algorithm

- Brute-force algorithm, which is also called the "naïve" is the simplest algorithm that can be used in pattern searching.
- In Brute Force Pattern Matching algorithm scanning from left to right.
- Let us consider a Text string of length n and Pattern string of length m.
- In Brute Force Pattern Matching algorithm pattern and text are compared character by character.
- If the Pattern character is found, hence compare immediate right character.
- If the Pattern character is not found, then shift pattern by one position to right.
- This makes the Brute Force Pattern Matching Algorithm to take more time and effort.

Brute Force Pattern Matching Algorithm

```
Algorithm BruteForce ( t[ 0...n-1], p[0....m-1] )
{
    for i = 0 to n-m do
    {
        j = 0;
        while( j<m and p[i] = t[i+j] )
            j = j + 1;
        if(j=m)
          return i;  // it means all the characters have matched and hence the Pattern string is
                        found at position i.
    }
    return -1;
}
```

The algorithm works with two pointers; a "text pointer" i and a "pattern pointer" j. For all (n-m) possibly valid shifts, pattern and text are compared; while text and pattern characters are equal, the pattern pointer is incremented. If a mismatch occurs, i is incremented, j is reset to zero and the comparing process is restarted. In case a match is found, the algorithm returns the position of the pattern; if not, it returns not found message.

Features of Brute-Force Algorithem.

1 : Comparison done from left to right.

2 : It requires 2n text characters comparison.

3 : Time complexity is O(mn).

4 : Always shift the pattern by one position to right.

5 : No preprocessing phase.

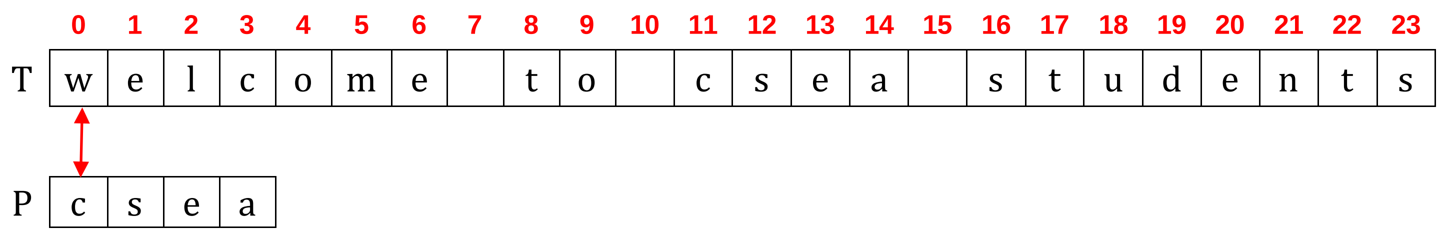
6 : it is slower process.

Example :

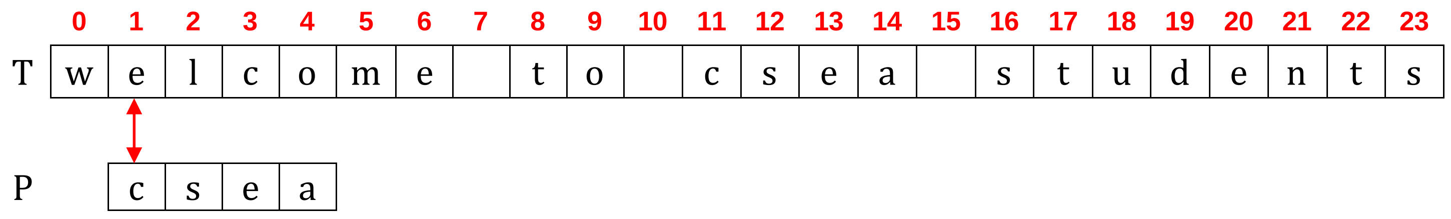
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
T	w	e	l	c	o	m	e		t	o		c	s	e	a		s	t	u	d	e	n	t	s

	0	1	2	3
P	c	s	e	a

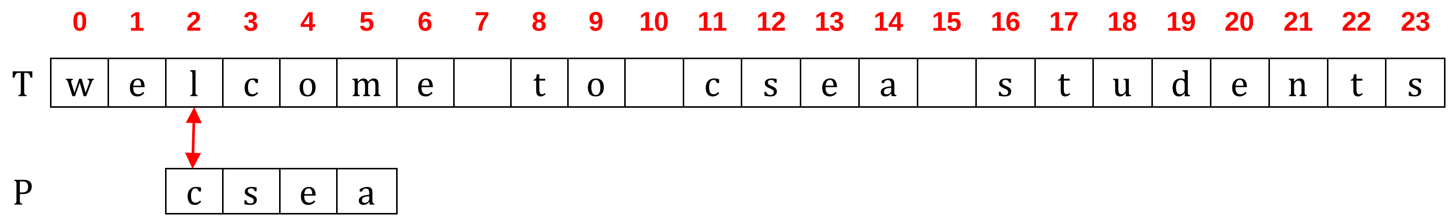
Step1 : Compare first character text and first character pattern, If match is not found, then shift pattern by one position to right.



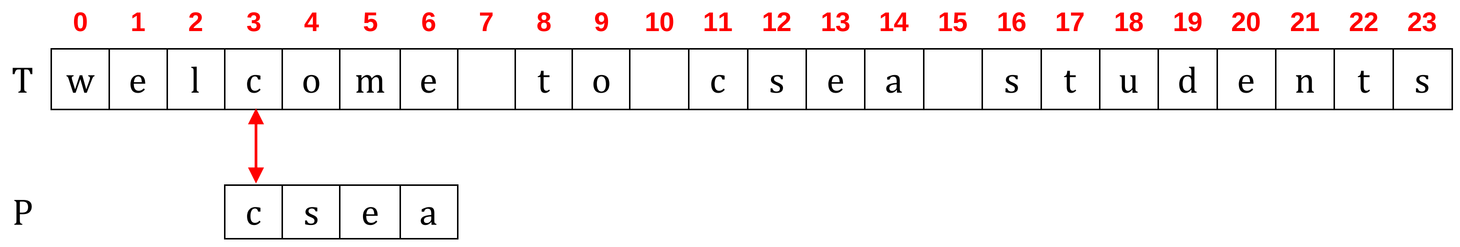
Step2 : Compare text and pattern characters, If match is not found, then shift pattern by one position to right.



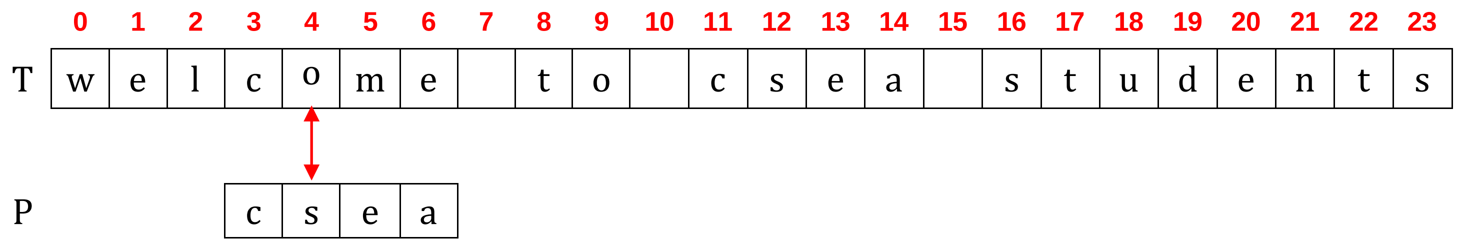
Step 3 : Compare text and pattern characters, If match is not found, then shift pattern by one position to right.



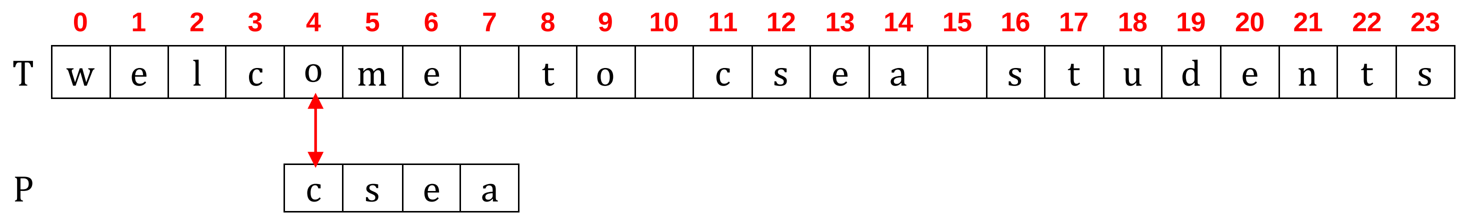
Step 4 : Compare text and pattern characters, If match is found, hence compare immediate right character.



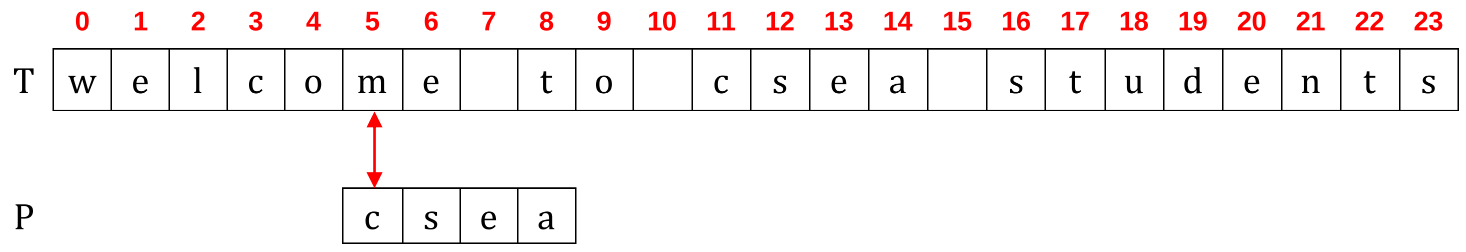
Step 5 : Compare text and pattern characters, If match is not found, then shift pattern by one position to right.



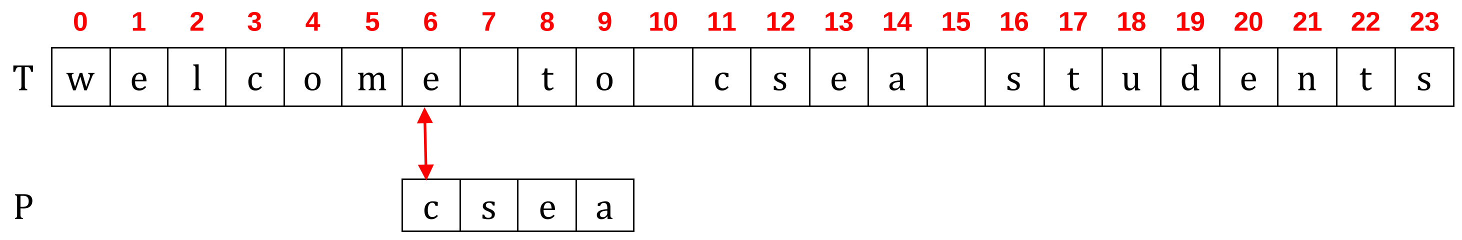
Step 6 : Compare text and pattern characters, If match is not found, then shift pattern by one position to right.



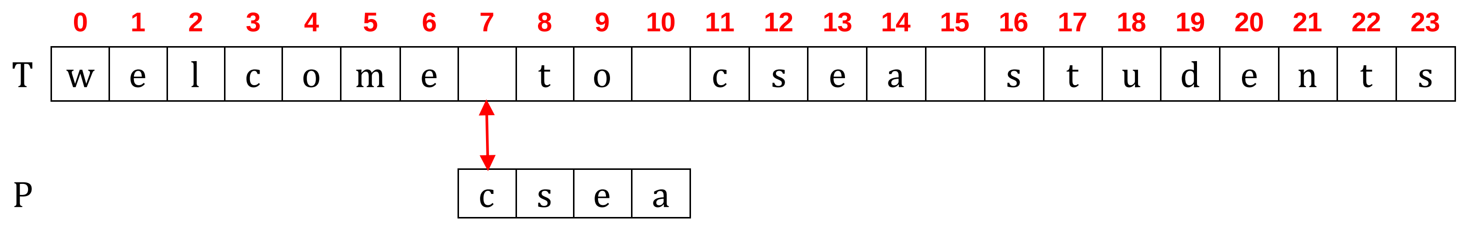
Step 7 : Compare text and pattern characters, If match is not found, then shift pattern by one position to right.



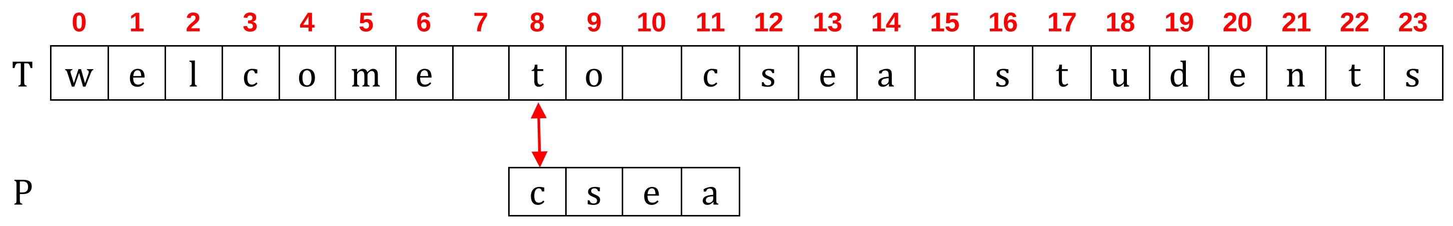
Step 8 : Compare text and pattern characters, If match is not found, then shift pattern by one position to right.



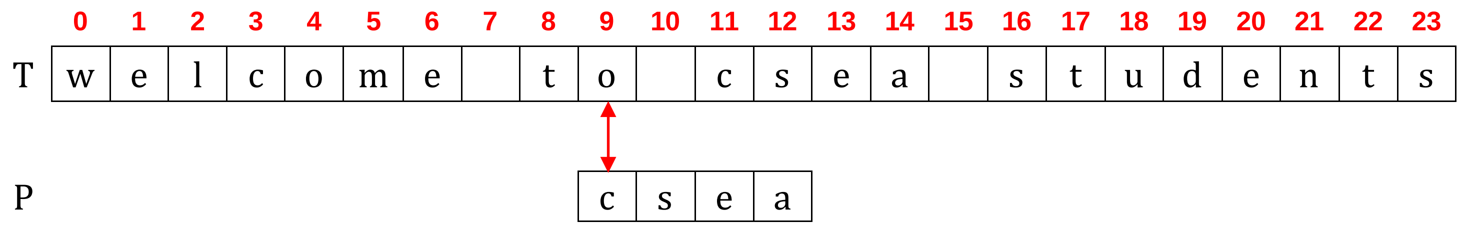
Step 9 : Compare text and pattern characters, If match is not found, then shift pattern by one position to right.



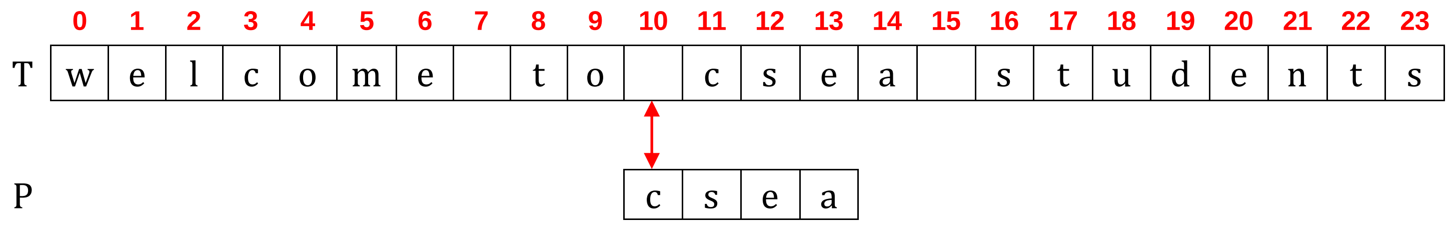
Step 10 : Compare text and pattern characters, If match is not found, then shift pattern by one position to right.



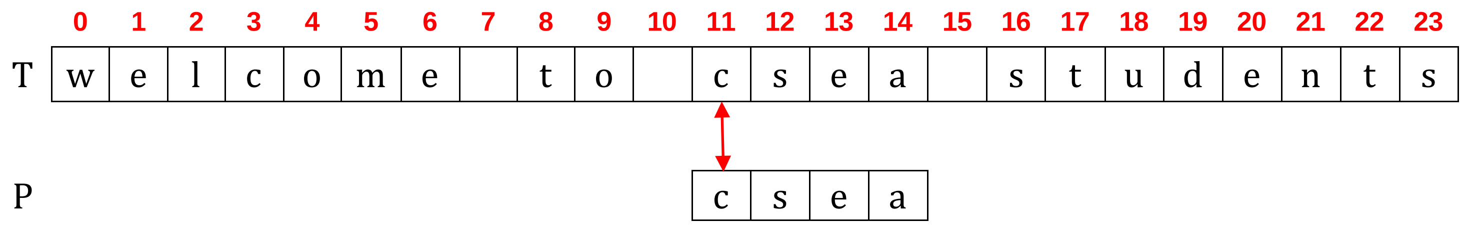
Step 11 : Compare text and pattern characters, If match is not found, then shift pattern by one position to right.



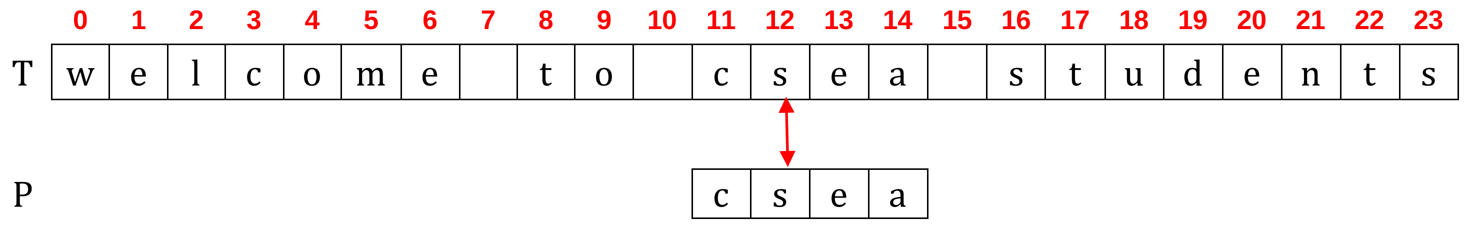
Step 12 : Compare text and pattern characters, If match is not found, then shift pattern by one position to right.



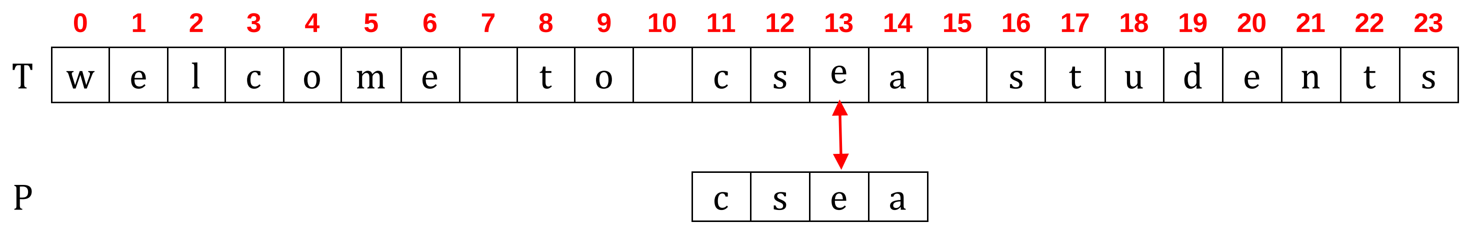
Step 12 : Compare text and pattern characters, If match is found, hence compare immediate right character.



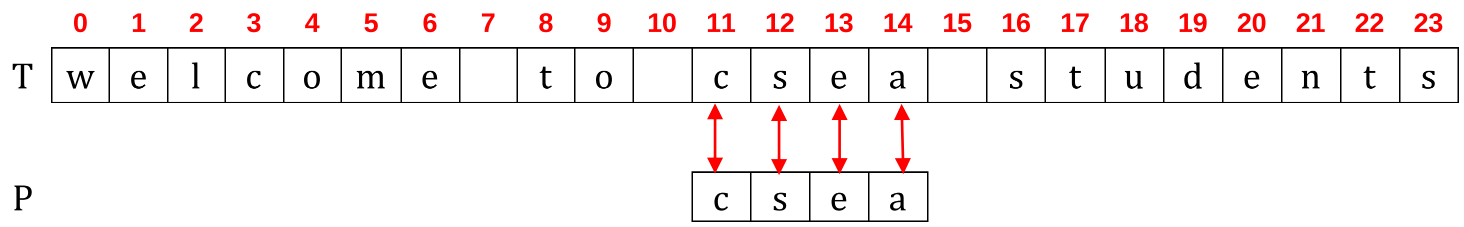
Step 13 : Compare text and pattern characters, If match is found, hence compare immediate right character.



Step 14 : Compare text and pattern characters, If match is found, hence compare immediate right character.



Step 15 : Compare text and pattern characters, Match is found.



2. Boyer Moore Algorithm

- Boyer-Moore algorithm is consider the most efficient string-matching algorithm.
- Boyer-Moore algorithm scans the characters of the pattern from right to left i.e beginning with the rightmost character.
- Boyer Moore is a combination of following two approaches.
  1) Bad Character Approach
  2) Good Suffix Approach
- Both of the above Approach can also be used independently to search a pattern in a text.

Bad Character Approach: The character of the text which doesn't match with the current character of pattern is called the Bad Character Approach. Upon mismatch we shift the pattern until

- The mismatch become a match, If the mismatch occur then we see the Bad-Match table for shifting the pattern.
- Pattern P move past the mismatch character. If the mismatch occur and the mismatch character not available in the Bad-Match Table then we shift the whole pattern accordingly.

Construct Bad Match Table

Formula for constructing Bad match table.

Values =max (1, Length of pattern – index – 1)

Pattern: TEAMMAST

Index	0	1	2	3	4	5	6	7
pattern	T	E	A	M	M	A	S	T

Length = 8.

T = max(1, 8-0-1)
= max(1,7) = 7
E = max(1, 8-1-1)
= max(1,6) = 6
A = max(1, 8-2-1)
= max(1,5) = 5
M = max(1, 8-3-1)
= max(1,4) = 4
M = max(1, 8-4-1)
= max(1,3) = 3

A = max(1, 8-5-1)
= max(1,2) = 2
S = max(1, 8-6-1)
= max(1,1) = 1
T = max(1, 8-7-1)
= max(1,0) = 1

Bad Match Table:

Letters	values
T	7
E	6
A	2
M	3
S	1
*	8

Note : Any other letter is presented by '*' is taken equal value to length of string i.e 8 here. Length =8

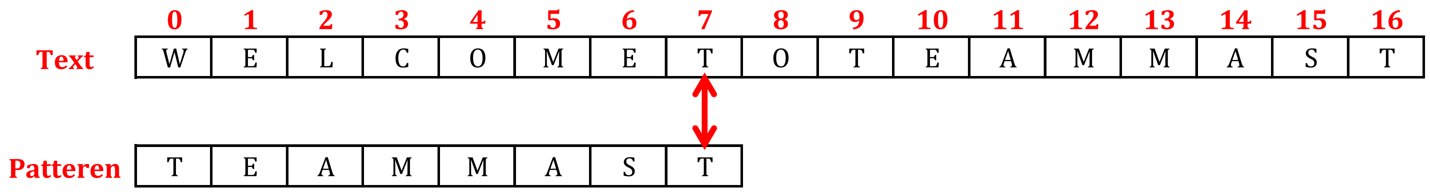
Boyer Moore Algorithm :

```
Boyer-Moore ( T [0...n ] , P [ 0...m ] )
  //set i and j to last index of P
      i ← m - 1
      j ← m – 1
  //loop to the end of the text string
  while i < n
      // if both characters match
      if P [ j ] = T [ i ]  then
          // and reached the end of P
          if j = 0 then
              //found a match
              return i
          else
              // go to next char
              i ← i - 1
              j ← j – 1
      else
       // skip over the whole word or shift to last occurrence
          i ← i + m - min ( j, 1 + last [ T [ i ] ] )
          j ← m – 1
  return -1
```

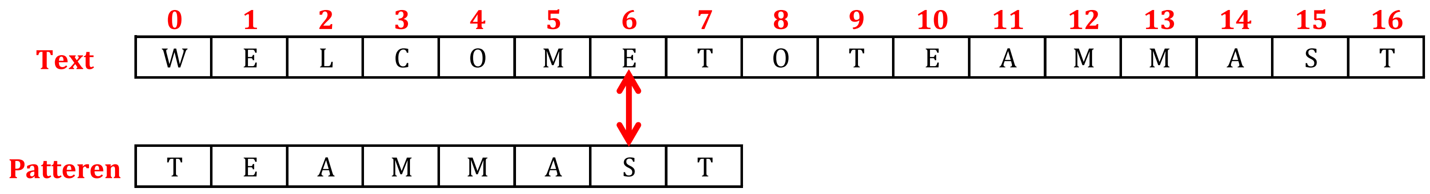
Example :

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Text	W	E	L	C	O	M	E	T	O	T	E	A	M	M	A	S	T
Pattern	T	E	A	M	M	A	S	T									

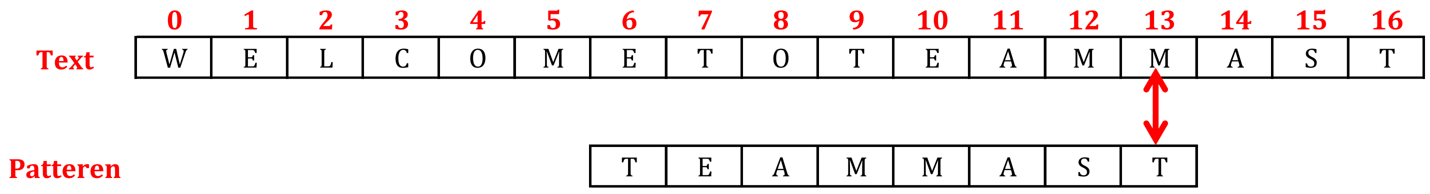
Step 1 : scans the characters of the pattern from right to left i.e beginning with the rightmost character. Compare text and pattern characters, If match is found, hence compare immediate left character.



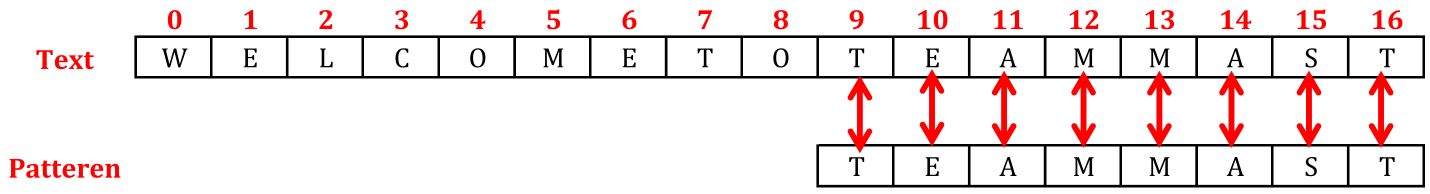
Step 2 : Compare text and pattern characters, If match is not found, we see the Bad-Match table for shifting the pattern. i.e, Move 6 spaces toward right.



Step 3 : Compare text and pattern characters, If match is not found, we see the Bad-Match table for shifting the pattern. i.e, Now move 3 spaces toward right.



Step 4 : Compare text and pattern characters, If match is found, hence compare immediate left character.



Match is found

3. Knuth-Morris-Pratt Algorithm

- KMP Algorithm is one of the most popular patterns matching algorithms.
- KMP stands for Knuth Morris Pratt.
- 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.
- Sometimes prefix table is also known as **LPS Table**.
- Here LPS stands for **"Longest proper Prefix which is also Suffix"**.

Steps for Creating LPS Table (Prefix Table)

Step 1 - Define a one dimensional array with the size equal to the length of the Pattern. (LPS[size])

Step 2 - Define variables **i & j**. Set i = 0, j = 1 and LPS[0] = 0.

Step 3 - Compare the characters at **Pattern[i]** and **Pattern[j].**

Step 4 - If both are matched then set **LPS[j] = i+1** and increment both i & j values by one. Goto to Step 3.

Step 5 - If both are not matched then check the value of variable 'i'. If it is '0' then set **LPS[j] = 0** and increment 'j' value by one, if it is not '0' then set **i = LPS[i-1]**. Goto Step 3.

Step 6- Repeat above steps until all the values of LPS[] are filled.

Example for creating KMP Algorithm's LPS Table (Prefix Table)

Consider the following Pattern

	0	1	2	3	4	5	6
Pattern :	A	B	C	D	A	B	D

Let us define LPS[] table with size 7 which is equal to length of the Pattern

	0	1	2	3	4	5	6
LPS							

Step 1 - Define variables i & j. Set i = 0, j = 1 and LPS[0] = 0.

	0	1	2	3	4	5	6
LPS	0						

i = 0 and j = 1

Step 2 - Campare Pattern[i] with Pattern[j] ===> A with B.
Since both are not matching and also "i = 0", we need to set LPS[j] = 0 and increment 'j' value by one.

	0	1	2	3	4	5	6
LPS	0	0					

i = 0 and j = 2

Step 3 - Campare Pattern[i] with Pattern[j] ===> A with C.
Since both are not matching and also "i = 0", we need to set LPS[j] = 0 and increment 'j' value by one.

	0	1	2	3	4	5	6
LPS	0	0	0				

i = 0 and j = 3

Step 4 - Campare Pattern[i] with Pattern[j] ===> A with D.
Since both are not matching and also "i = 0", we need to set LPS[j] = 0 and increment 'j' value by one.

	0	1	2	3	4	5	6
LPS	0	0	0	0			

i = 0 and j = 4

Step 5 - Campare Pattern[i] with Pattern[j] ===> A with A.
Since both are matching set LPS[j] = i+1 and increment both i & j value by one.

	0	1	2	3	4	5	6
LPS	0	0	0	0	1		

i = 1 and j = 5

Step 6 - Campare Pattern[i] with Pattern[j] ===> B with B.
Since both are matching set LPS[j] = i+1 and increment both i & j value by one.

	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	

i = 2 and j = 6

Step 7 - Campare Pattern[i] with Pattern[j] ===> C with D.
Since both are not matching and i !=0, we need to set i = LPS[i-1]
===> i = LPS[2-1] = LPS[1] = 0.

	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	

i = 0 and j = 6

Step 8 - Campare Pattern[i] with Pattern[j] ===> A with D.
Since both are not matching and also "i = 0", we need to set LPS[j] = 0 and increment 'j' value by one.

	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	0

Here LPS[] is filled with all values so we stop the process. The final LPS[] table is as follows...

	0	1	2	3	4	5	6
LPS	**0**	**0**	**0**	**0**	**1**	**2**	**0**

How to use LPS Table

- We use the LPS table to decide how many characters are to be skipped for comparison when a mismatch has occurred.
- When a mismatch occurs, check the LPS value of the previous character of the mismatched character in the pattern.
- If it is '0' then start comparing the first character of the pattern with the next character to the mismatched character in the text.
- If it is not '0' then start comparing the character which is at an index value equal to the LPS value of the previous character to the mismatched character in pattern with the mismatched character in the Text.

Example : KMP A lgorithm

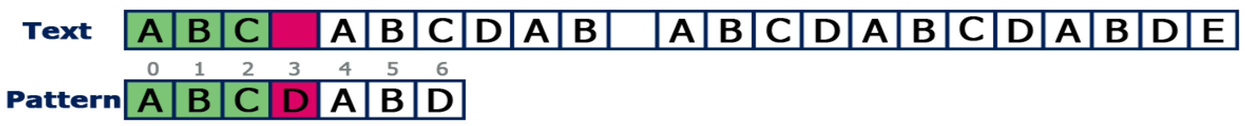
Consider the following Text and Pattern

Text : ABC ABCDAB ABCDABCDABDE
Pattern : ABCDABD

LPS[] table for the above pattern is as follows...

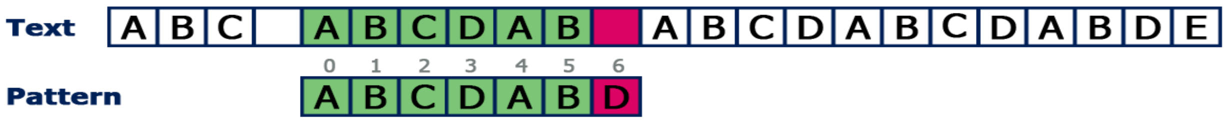
	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	0

Step 1 - Start comparing first character of Pattern with first character of Text from left to right



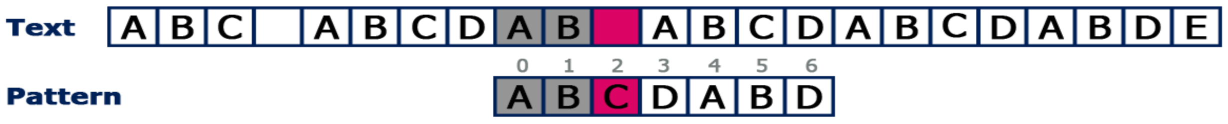
Here mismatch occured at Pattern[3], so we need to consider LPS[2] value. Since LPS[2] value is '0' we must compare first character in Pattern with next character in Text.

Step 2 - Start comparing first character of Pattern with next character of Text.



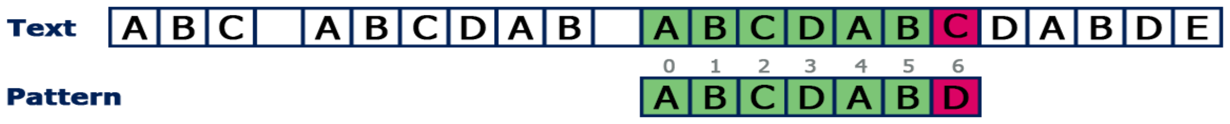
Here mismatch occured at Pattern[6], so we need to consider LPS[5] value. Since LPS[5] value is '2' we compare Pattern[2] character with mismatched character in Text.

Step 3 - Since LPS value is '2' no need to compare Pattern[0] & Pattern[1] values



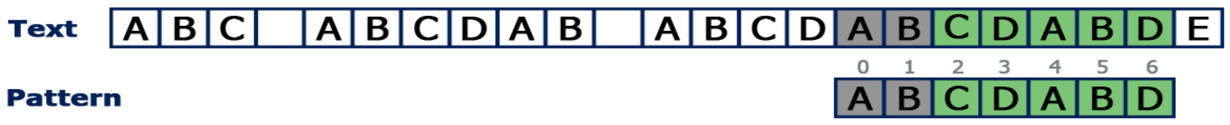
Here mismatch occured at Pattern[2], so we need to consider LPS[1] value. Since LPS[1] value is '0' we must compare first character in Pattern with next character in Text.

Step 4 - Compare Pattern[0] with next character in Text.



Here mismatch occured at Pattern[6], so we need to consider LPS[5] value. Since LPS[5] value is '2' we compare Pattern[2] character with mismatched character in Text.

Step 5 - Compare Pattern[2] with mismatched character in Text.



Here all the characters of Pattern matched with a substring in Text which is starting from index value 15. So we conclude that given Pattern found at index 15 in Text.

Tries :

- A trie is a multiway tree data structure used for storing strings over an alphabet.
- It is used to store a large amount of strings. The pattern matching can be done efficiently using tries.
- Trie is also called as Prefix Tree and some times Digital Tree.

Different Types of Tries

1 : Standard Trie

2 : Compressed Trie.

3 : Suffix Trie.

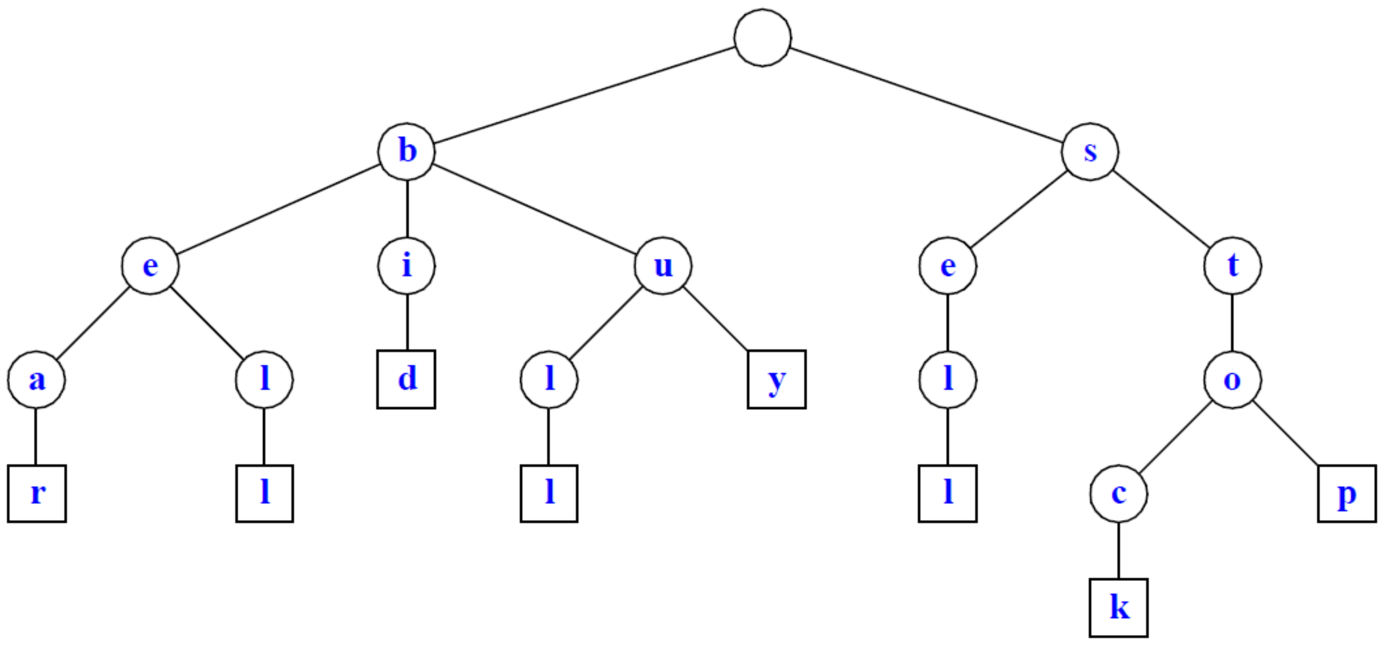
1 : Standard Trie : The standard trie for a set of strings S is an ordered tree such that

1 : Each node is labeled with an alphabet except root node.

2 : The children of a node are alphabetically ordered.

3 : The path from root to external node should produce a string from set S.

- Example: standard trie for the set of strings
  S = { bear, bell, bid, bull, buy, sell, stock, stop }



A standard trie uses O(n) space. Operations (find, insert, remove) take time O(dm) each,

where:

n = total size of the strings in S,

m =size of the string parameter of the operation

d =alphabet size

Applications of Standard Tries :

- A standard trie supports the following operations on a preprocessed text in time O(m), where m = |X|

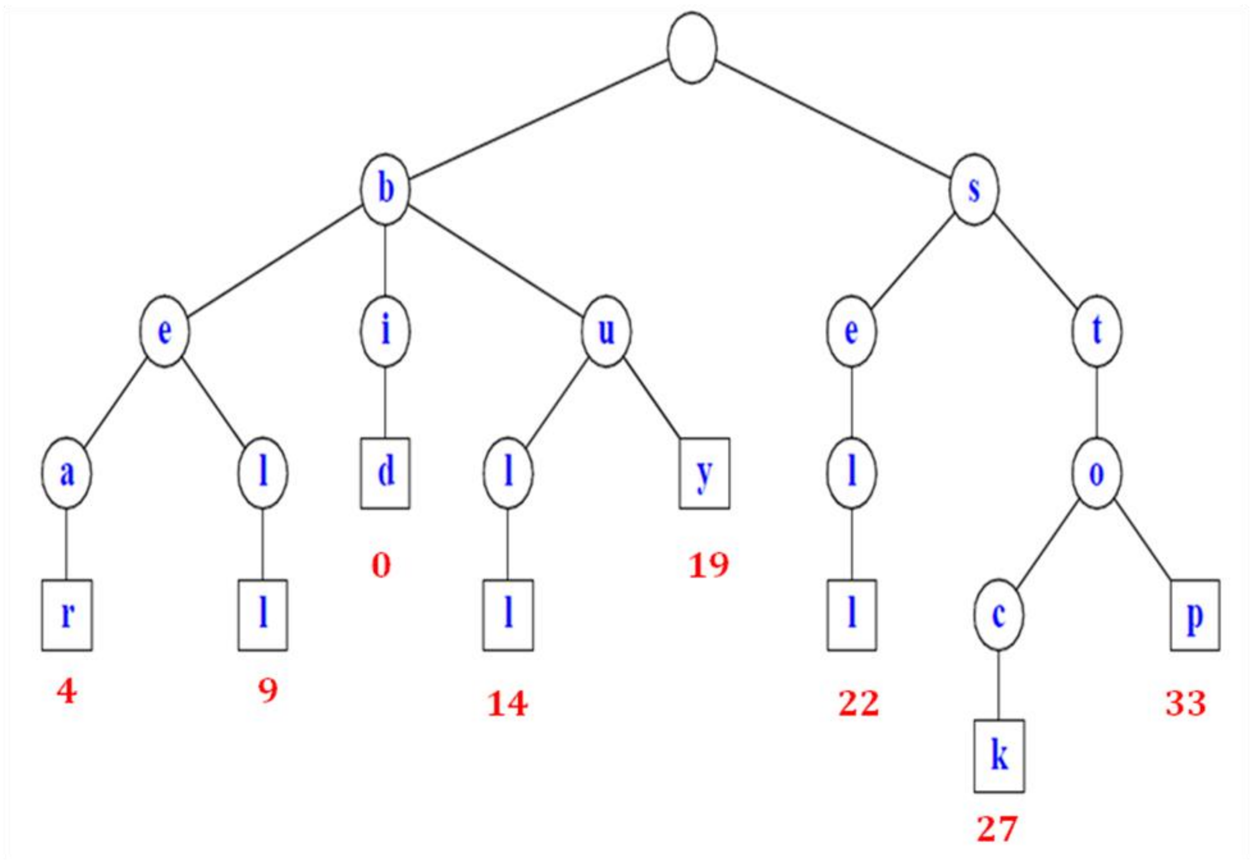
  word matching: find the first occurrence of word X in the text

  prefix matching: find the first occurrence of the longest prefix of word X in the text
- Each operation is performed by tracing a path in the trie starting at the root.

Example : Consider a text

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
b	i	d		b	e	a	r		b	e	l	l		b	u	l	l		b	u	y

22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
s	e	l	l		s	t	o	c	k		s	t	o	p



Compressed Trie :

- Compressed trie is a trie in which each internal node has greater than or equal to two child.

  Internal node >= 2 child nodes

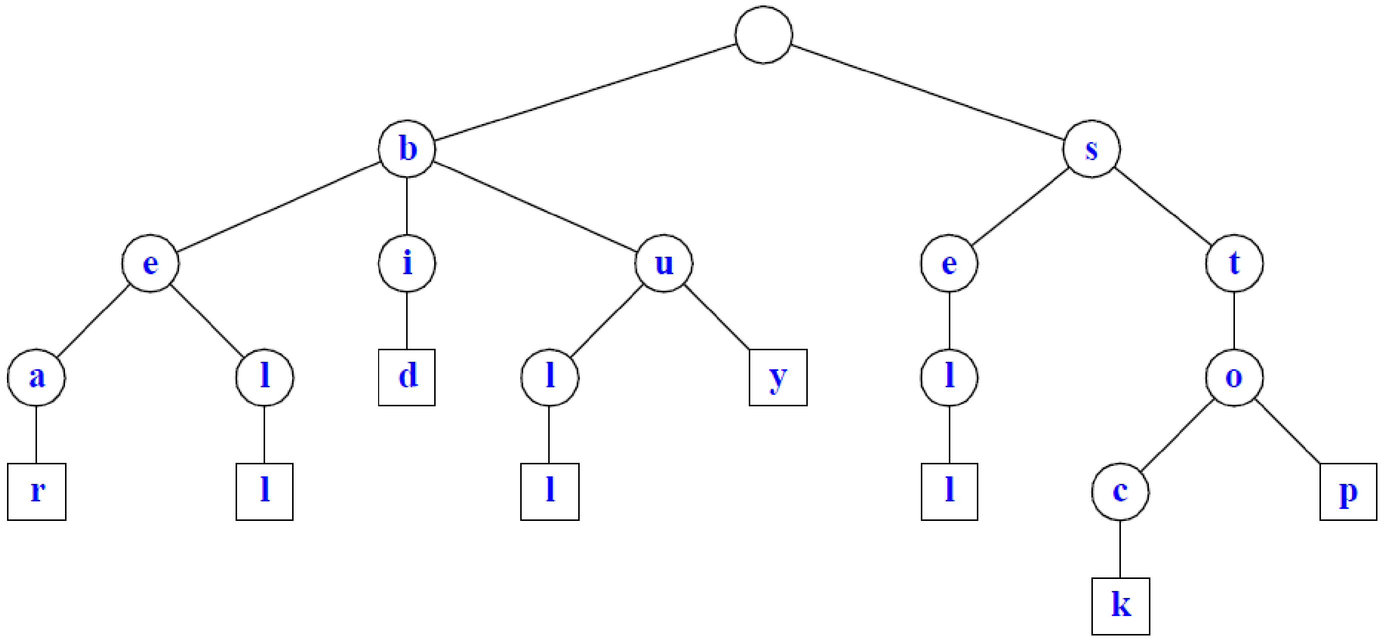
- Node may be labeled with multiple characters (String)
- It is obtained from standard trie by compressing chain of "redundant " nodes.
- Redundant Node means
- Internal node is redundant if

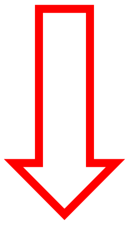
  1. Node is not the root node
  2. Node has one child.

Standared Trie :

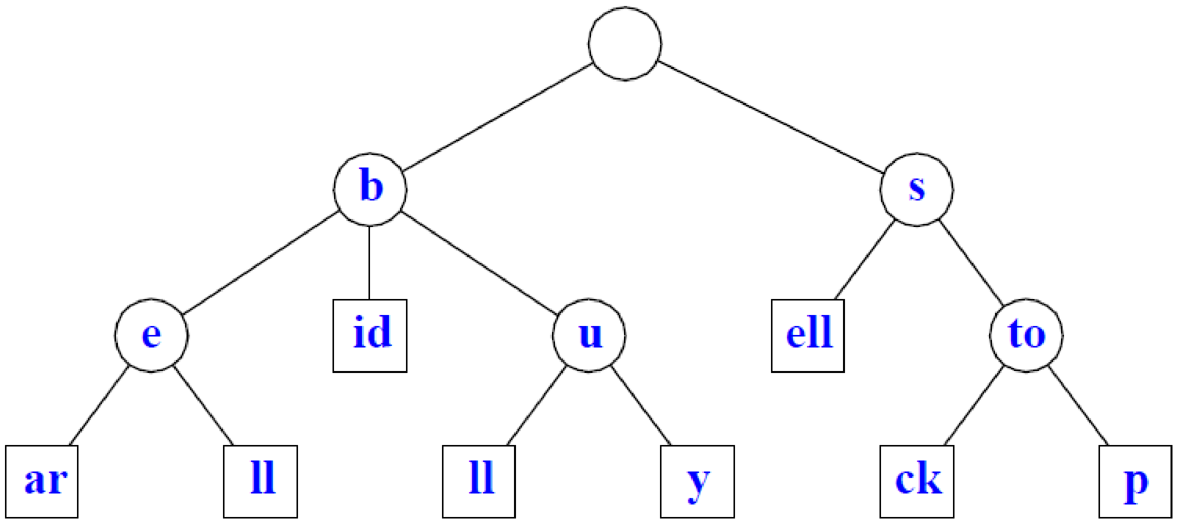
Example :

s = { bear, bell, bid, bull, buy, sell, stock, stop }





Compressed Trie



Representation of Compressed Trie

In this representation, each node 3 parameters, i.e, i , j , k

Where,

i = Index of String

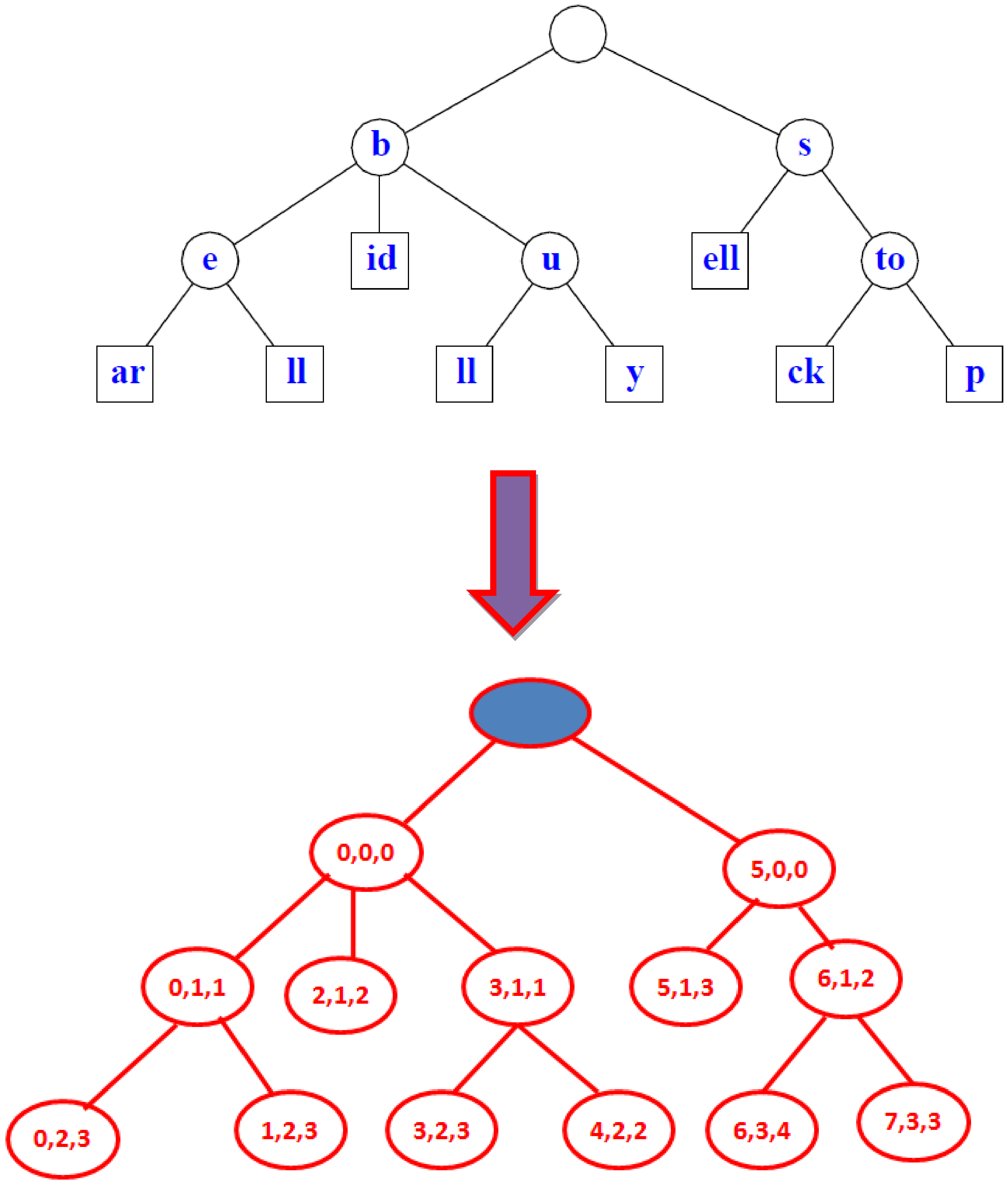
j = Starting position of the character

k = Ending position of the character

s = { bear, bell, bid, bull, buy, sell, stock, stop }

	0	1	2	3	4
S[0]	b	e	a	r	
S[1]	b	e	l	l	
S[2]	b	i	d		
S[3]	b	u	l	l	
S[4]	b	u	y		
S[5]	s	e	l	l	
S[6]	s	t	o	c	k
S[7]	s	t	o	p	

Compressed Trie



Suffix Tries : Suffix trie is compressed trie for all the suffixies of a text.

Example : Consider the given text.

Text = "banana"

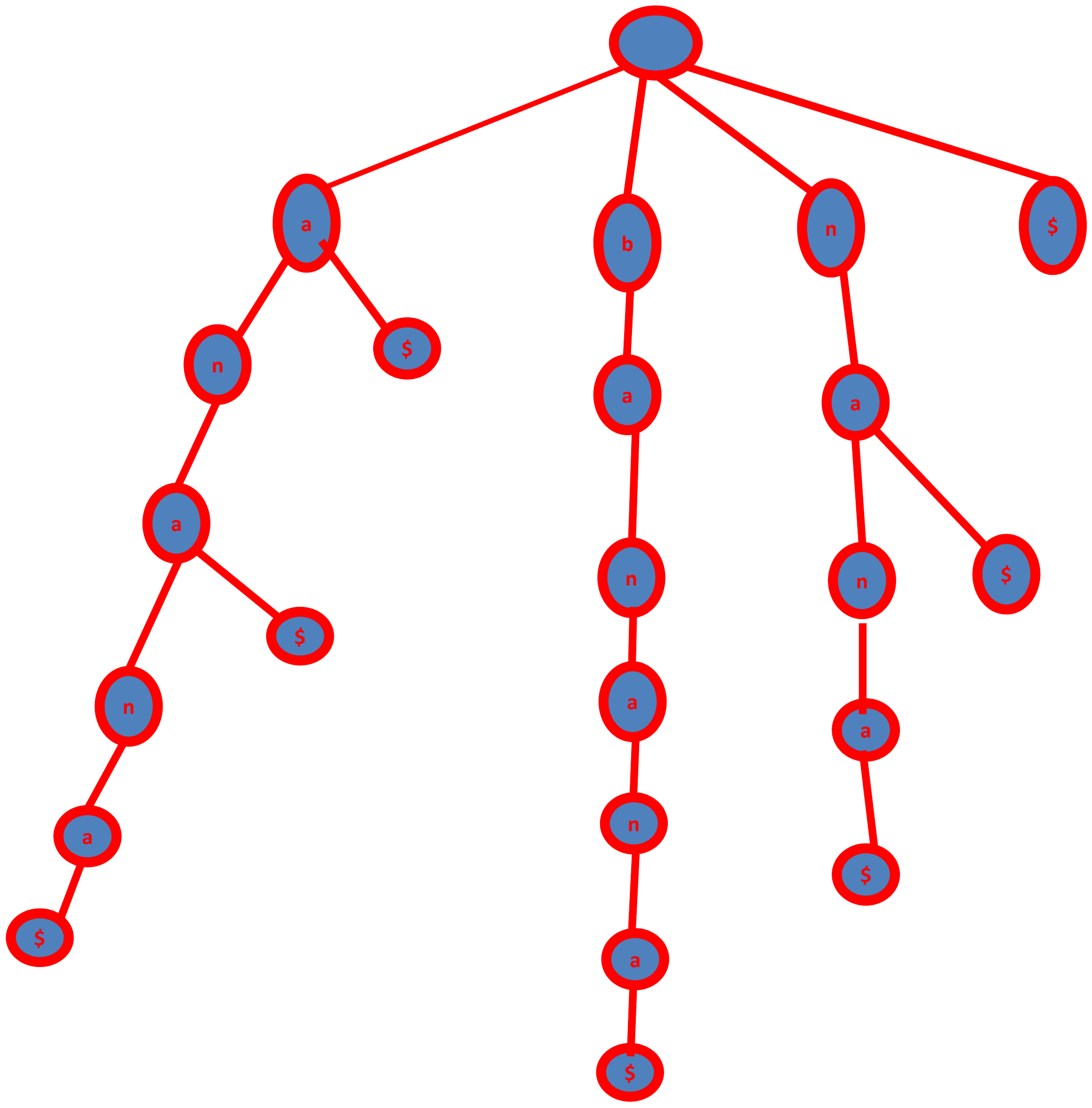
Step1 : Append special character "$" at the end of text.

Text = "banana$"

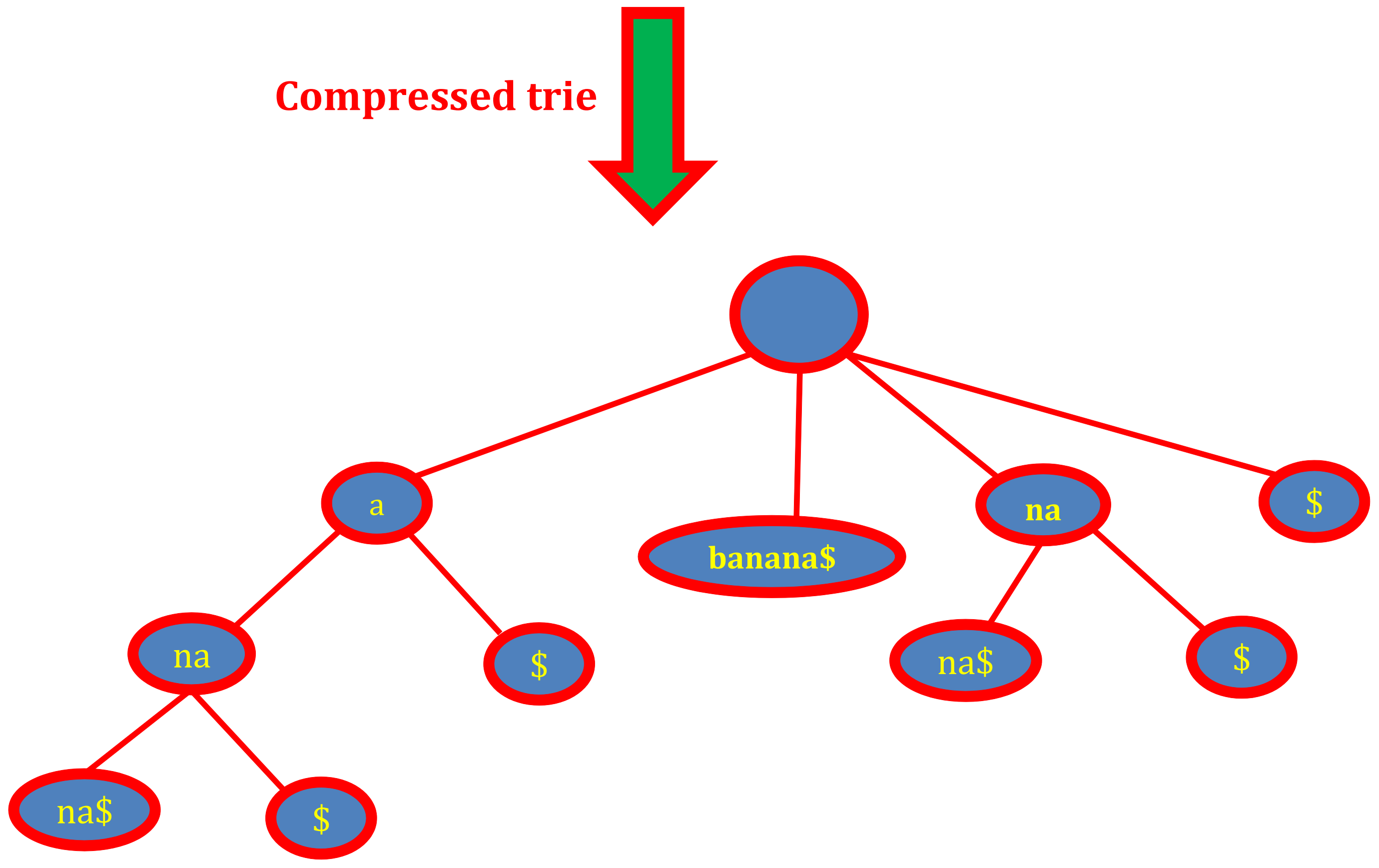
Step2 : Generate all suffix for the given text.

```
b  a  n  a  n  a  $
   a  n  a  n  a  $
      n  a  n  a  $
         a  n  a  $
            n  a  $
               a  $
                  $
```

Standared Trie for the above suffixes.

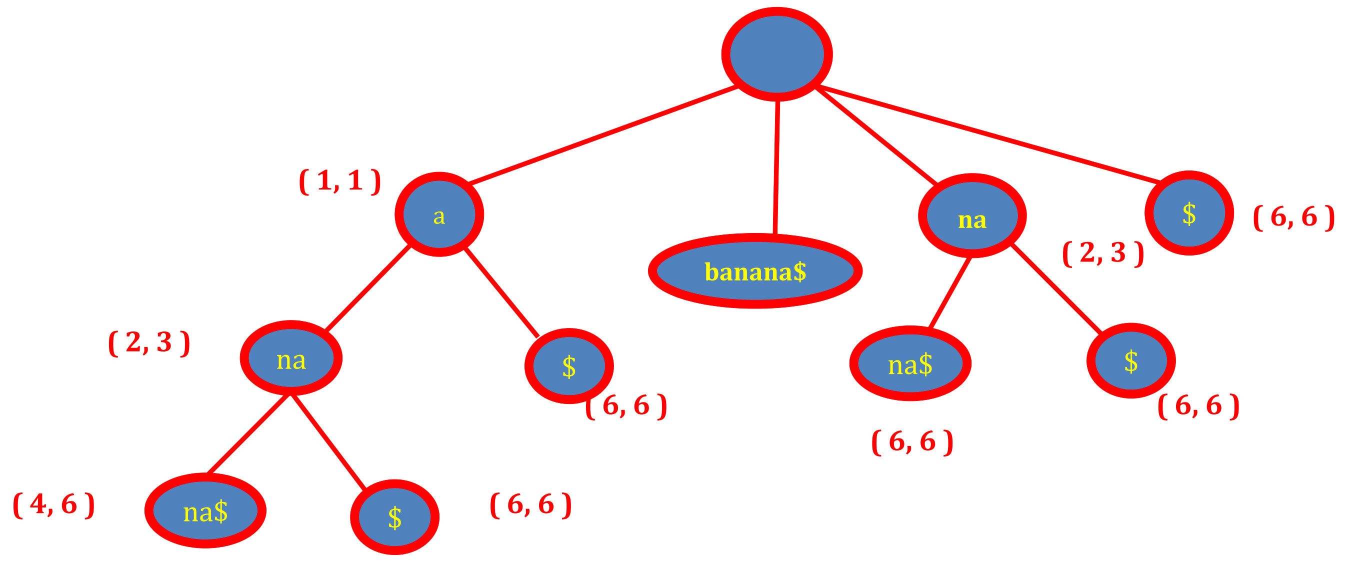


Compressed trie



Representation of Suffix Trie

0	1	2	3	4	5	6
b	a	n	a	n	a	$


(1, 1)
a
banana$
na
$
(6, 6)
(2, 3)
(2, 3)
na
$
(6, 6)
na$
$
(6, 6)
(6, 6)
(4, 6)
na$
$
(6, 6)