CS F211: DATA STRUCTURES & ALGORITHMS (2ND SEMESTER 2024-25) PATTERN MATCHING ALGORITHMS

Chittaranjan Hota, PhD
Senior Professor, Computer Sc.
BITS-Pilani Hyderabad Campus
hota[AT]hyderabad.bits-pilani.ac.in

PATTERN MATCHING: WHAT IS & WHY?

- Pattern matching is a programming technique used to check whether a given sequence of data (such as a string, or a list) follows a specific pattern.
- Searching: Imagine you are using ctrl+F in browser or a document to find out all occurrences of a word “climate” in a long document.
- Behind the scene, either Boyer- Moore or KMP might be working.

DNA Sequence = "ACGTTATGCGTACGATGCGATACG" DNA Pattern = "ATGCG" → [5, 15]

- Antivirus tools (Norton, McAfee, Avast) looking for presence of malicious signatures in an infected file.

File content: "90906A2B68576133FFD26A2B68576190" Pattern = "6A2B685761" → ⚠ [4,20]

- Email spam: Congratulations! You have been selected as the lucky winner of our INR 3.0 crores Mega Prize. To claim your reward, please send us your full name, address, and banking details immediately.
- Ecommerce: “red running shoes” → “Nike red running shoes for Men” (Basic text search), “red runing shoes” → “Nike red running shoes” (Fuzzy matching), “shoes for jogging” → “running shoes”, “sports shoes” (Semantic search).

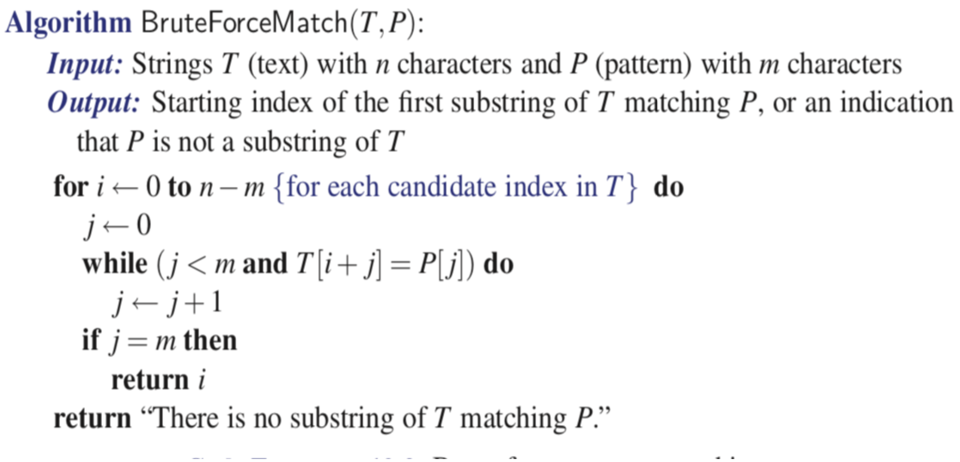
STRING PROCESSING

- A common problem in text editing, DNA sequence analysis, and web crawling: finding strings inside other strings.

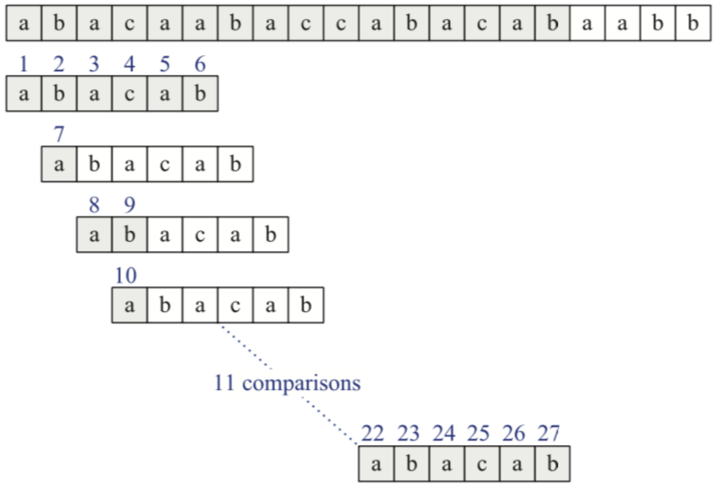
- *P* = “CGTAAACTGCTTTAATCAAACGC”
- *S* = “http://www.wiley.com”

- Whether “TTTAA” is a substring of the above sequence?

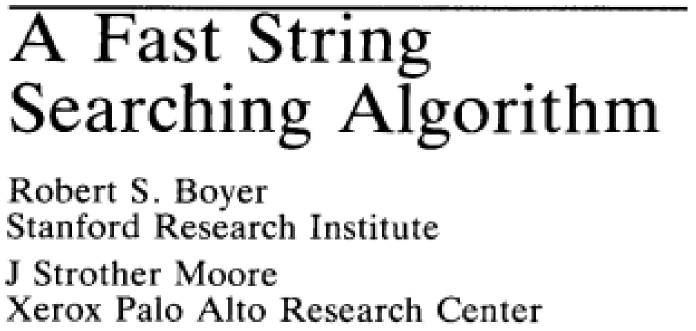
O((n-m+1)m) ➔ O(nm)



```
Algorithm BruteForceMatch($T,P$):
   Input: Strings $T$ (text) with $n$ characters and $P$ (pattern) with $m$ characters
   Output: Starting index of the first substring of $T$ matching $P$, or an indication
      that $P$ is not a substring of $T$
   for $i \leftarrow 0$ to $n-m$ {for each candidate index in $T$} do
      $j \leftarrow 0$
      while ($j < m$ and $T[i+j] = P[j]$) do
         $j \leftarrow j+1$
      if $j = m$ then
         return $i$
   return “There is no substring of $T$ matching $P$.”
```



ACM, 1977



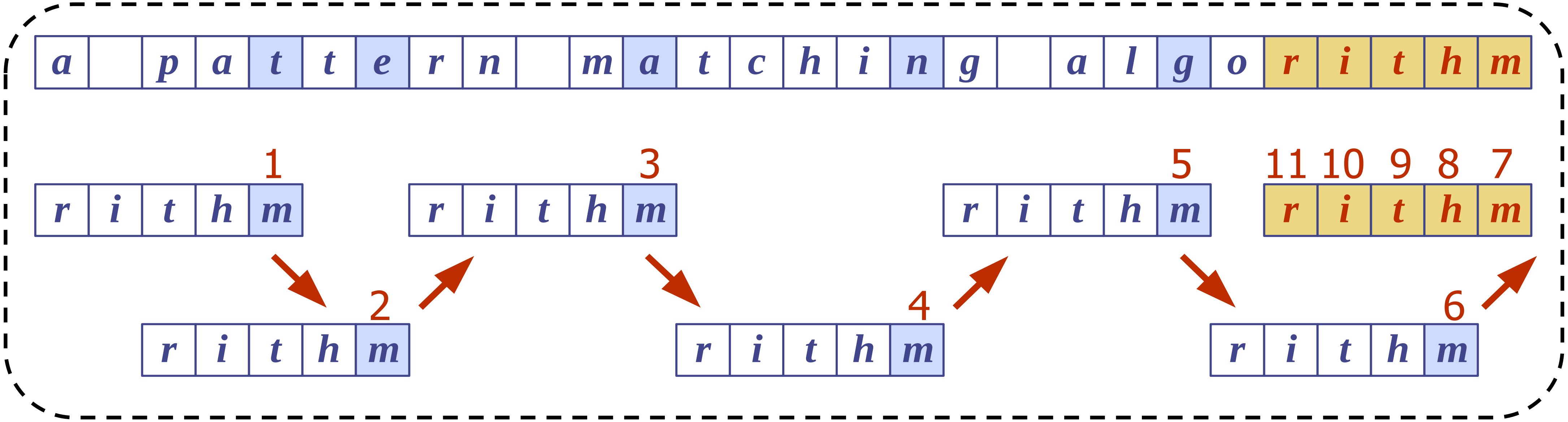
A Fast String Searching Algorithm

Robert S. Boyer
Stanford Research Institute
J Strother Moore
Xerox Palo Alto Research Center

BOYER-MOORE ALGORITHM

- Efficient string-searching algorithms used in pattern matching, especially when the pattern is relatively short compared to the text
- Looking-glass heuristic: Right to left. Purpose is to find mismatches early.
- Character-jump (Bad character) heuristic: Shift pattern smartly. Purpose is to avoid unnecessary comparisons or skips many comparisions.

EXAMPLE



a p a t t e r n m a t c h i n g a l g o r i t h m

1 3 5 11 10 9 8 7

r i t h m r i t h m r i t h m r i t h m

2 4 6

r i t h m r i t h m r i t h m

THE BOYER-MOORE ALGORITHM CONTINUED…

```
Algorithm BoyerMooreMatch(T, P, S)
    L ← lastOccurenceFunction(P, S )
    i ← m - 1
    j ← m - 1
    repeat
        if (T[i] == P[j])
            if  j == 0
                return  i  { match at i }
            else
                i ← i - 1
                j ← j - 1
        else
            { character-jump }
            l ← L[T[i]]
            i ← i + m – min(j, 1 + l)
            j ← m - 1
    until  i > n - 1
    return  -1 { no match }
```

T="BITSPILANI" → n=10

P="PILANI" → m=6

Create a table showing the rightmost occurrence of each character in the pattern (L):

P:

P	I	L	A	N	I
0	1	2	3	4	5

L:

P	I	L	A	N
0	5	2	3	4

Run of the algorithm:

i=4 i=5 i=9

T:	B	I	T	S	P	I	L	A	N	I
P:	P	I	L	A	N	I				

j=4 j=5

l=L[T[4]]=L[P]=0 → i = i +m – min(j,1+l) = 4+6-min(4,1+0)= 9

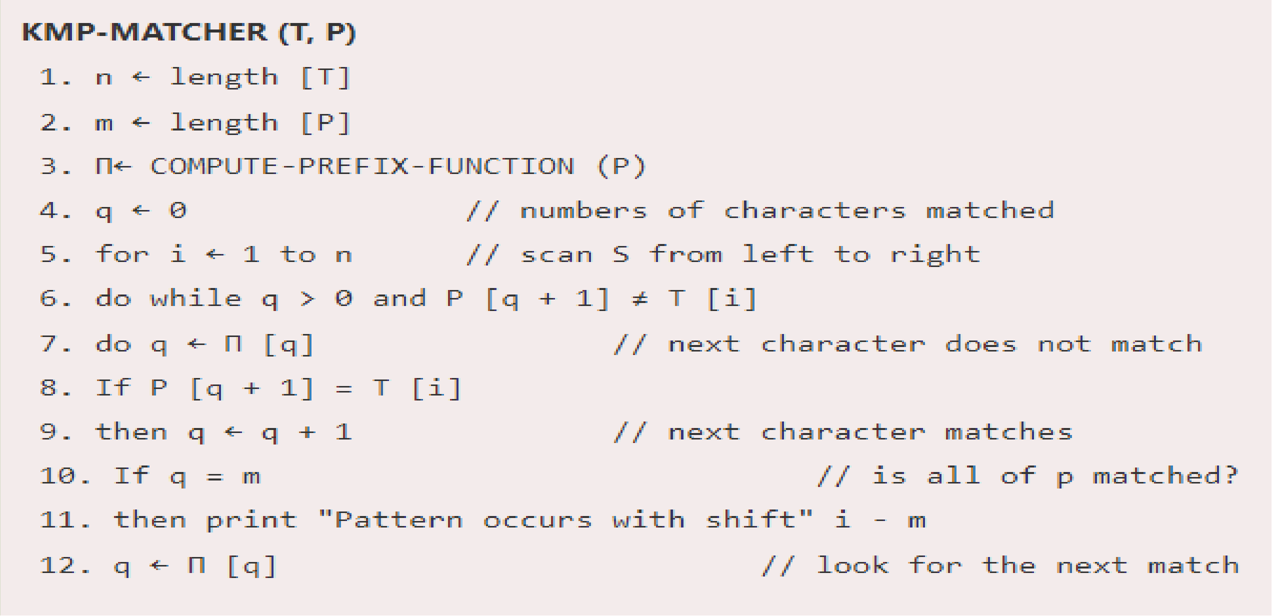
Repeat the loop with i=9, and j=m-1=5 → Finally, match found.

THE KNUTH-MORRIS-PRATT'S ALGORITHM

Knuth-Morris-Pratt's (KMP) algorithm compares the pattern to the text in left-to-right, but shifts the pattern more intelligently than the brute-force algorithm.

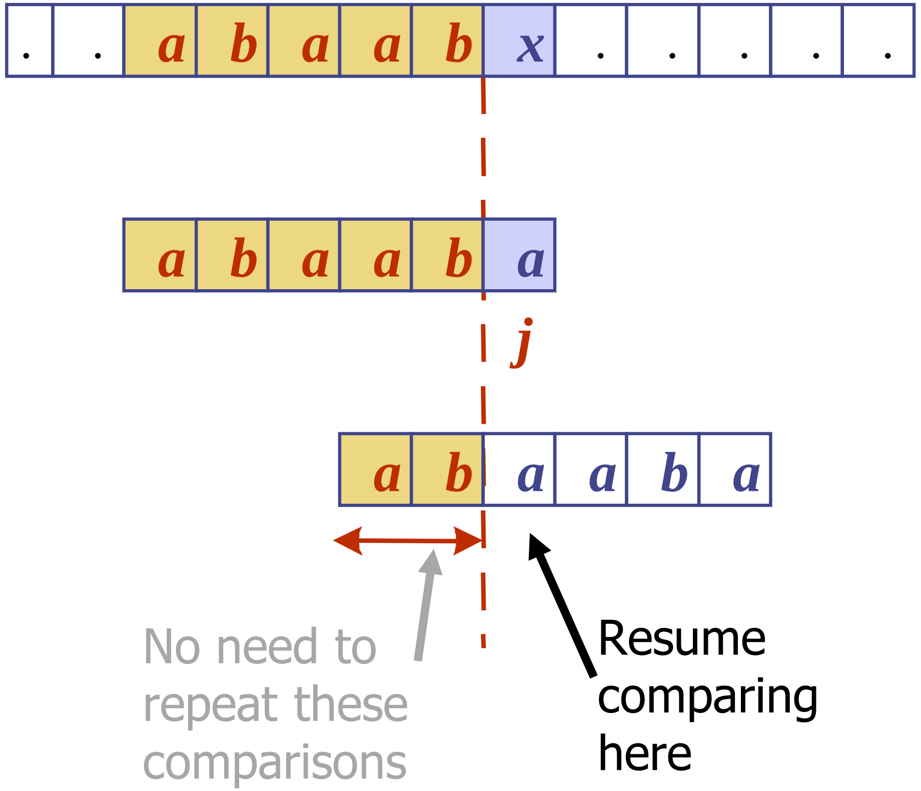
When a mismatch occurs, what is the most we can shift the pattern so as to avoid redundant comparisons?

Answer: the largest prefix of $P[0..j]$ that is a suffix of $P[1..j]$



KMP-MATCHER (T, P)

```
n ← length [T]
m ← length [P]
Π← COMPUTE-PREFIX-FUNCTION (P)
q ← 0                  // numbers of characters matched
for i ← 1 to n         // scan S from left to right
do while q > 0 and P [q + 1] ≠ T [i]
do q ← Π [q]                   // next character does not match
If P [q + 1] = T [i]
then q ← q + 1                 // next character matches
If q = m                                  // is all of p matched?
then print "Pattern occurs with shift" i - m
q ← Π [q]                             // look for the next match
```


. . a b a a b x . . . . .
a b a a b a
j
a b a a b a
No need to repeat these comparisons
Resume comparing here

EXAMPLE

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**

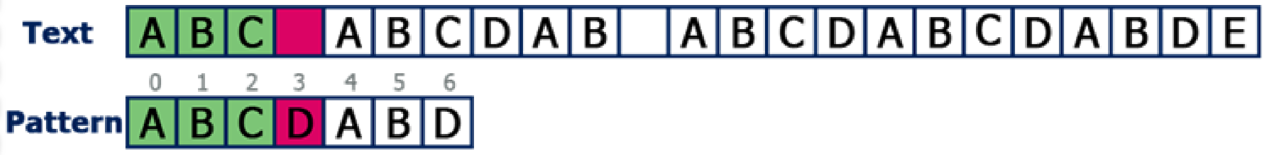
KMP Algo

Text:	A	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E
Pattern:	A	B	C	D	A	B	D																

LPS Table for the above Pattern is:

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

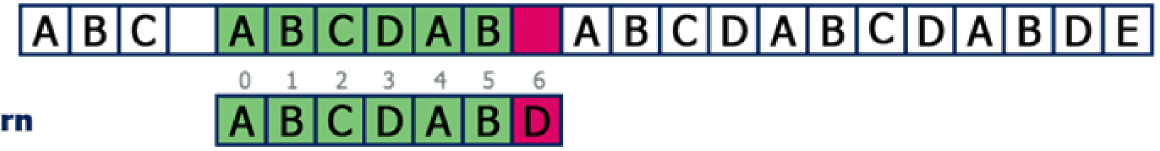
Step 1 - Start comparing first character of Pattern with first character of Text from left to right



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

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.

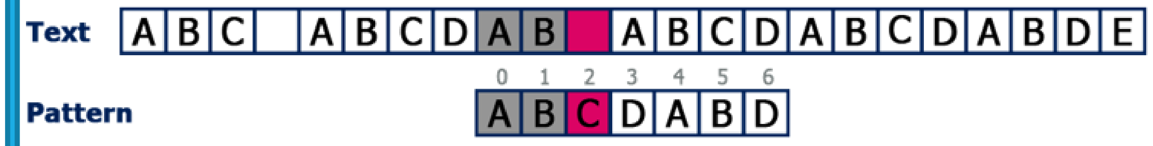
Step 2 - Start comparing first character of Pattern with next character of Text.



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

What should you do now?

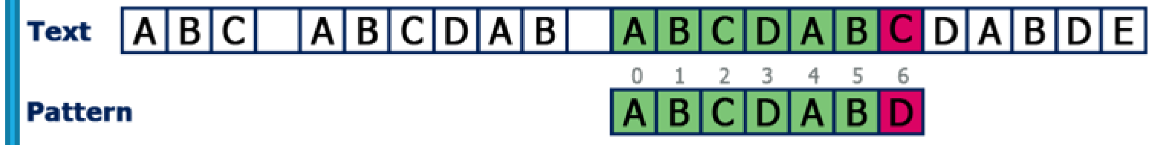
Step 3 - Since LPS value is '2' no need to compare Pattern[0] & Pattern[1] values



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

What should you do now?

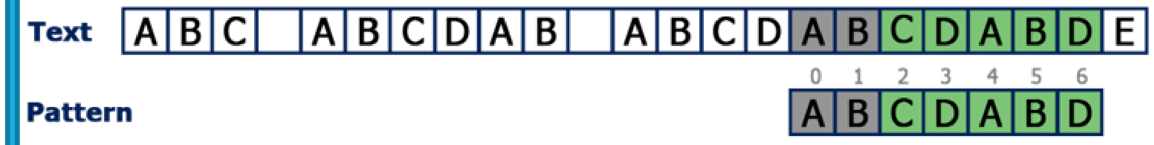
Step 4 - Compare Pattern[0] with next character in Text.



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

What should you do now?

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



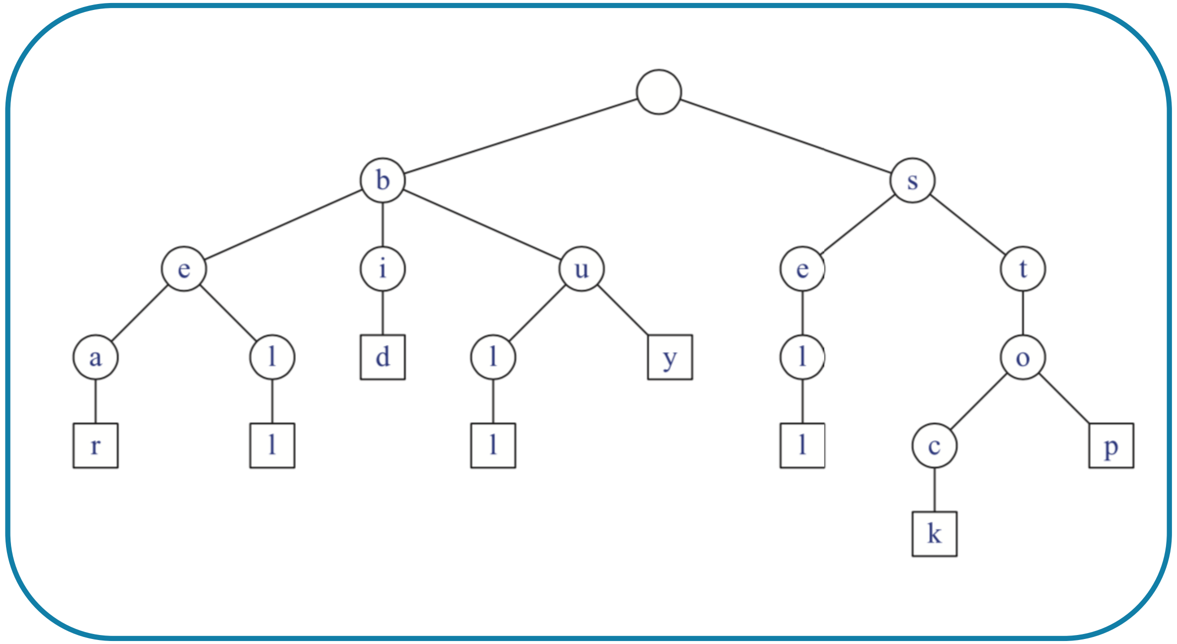
Text	A	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E
																0	1	2	3	4	5	6	
Pattern																A	B	C	D	A	B	D	

→ Pattern found at Index: 15 in the Text

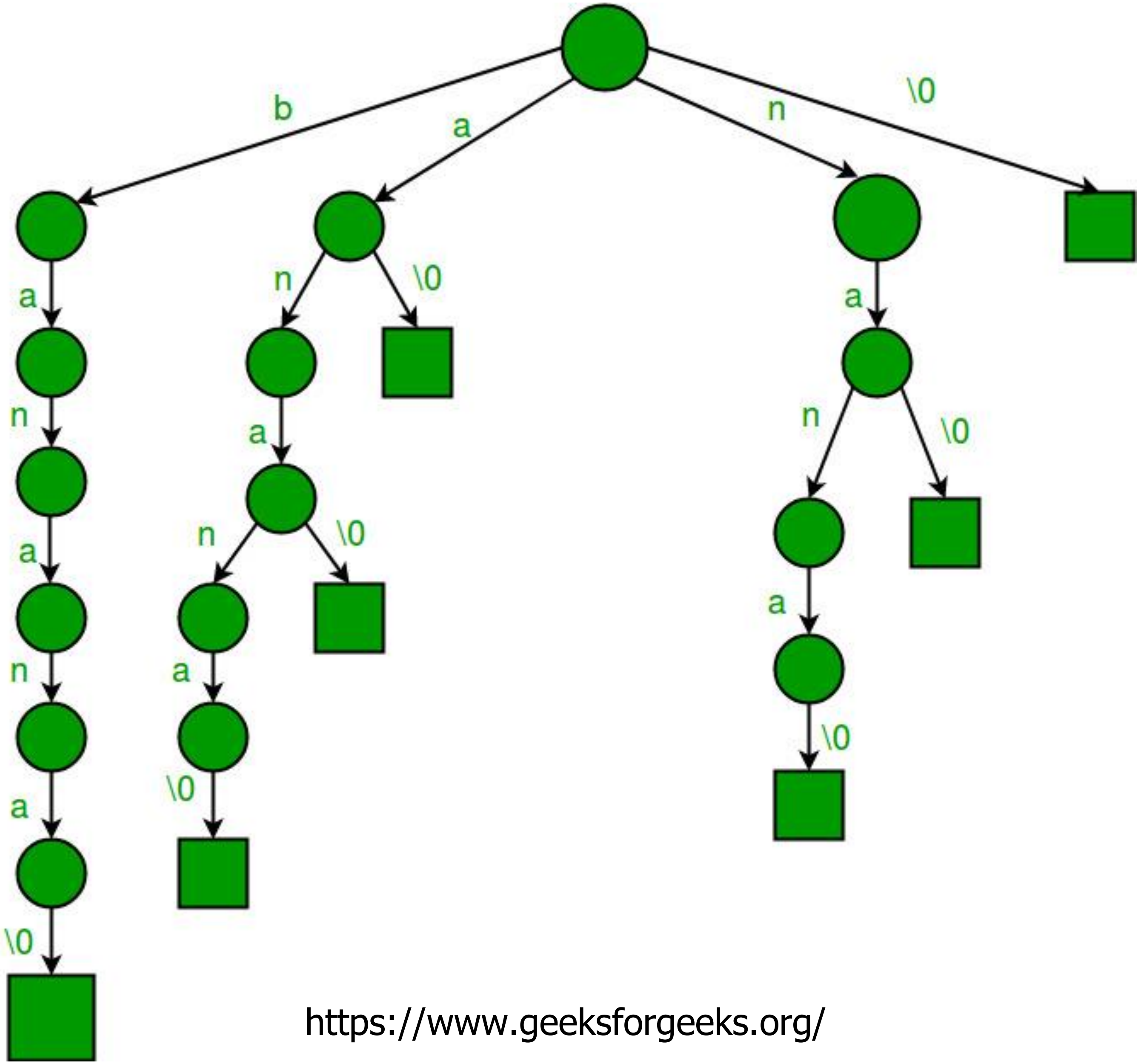
Tries: Efficient pattern matching

- Trie is a tree like data structure used to store collection of strings. Efficient in retrieval.
- A trie searches a string in O(m) time complexity, where **m** is the length of the string.
- In a trie, every node except the root stores a character value.
- All the children of a node are alphabetically ordered. If any two strings have a common prefix then they will have the same ancestors.

- We can model the set **S** (strings) as a rooted tree **T** in such a way, that each path from the root of **T** to any of its nodes, corresponds to a prefix of at least one string of **S**.



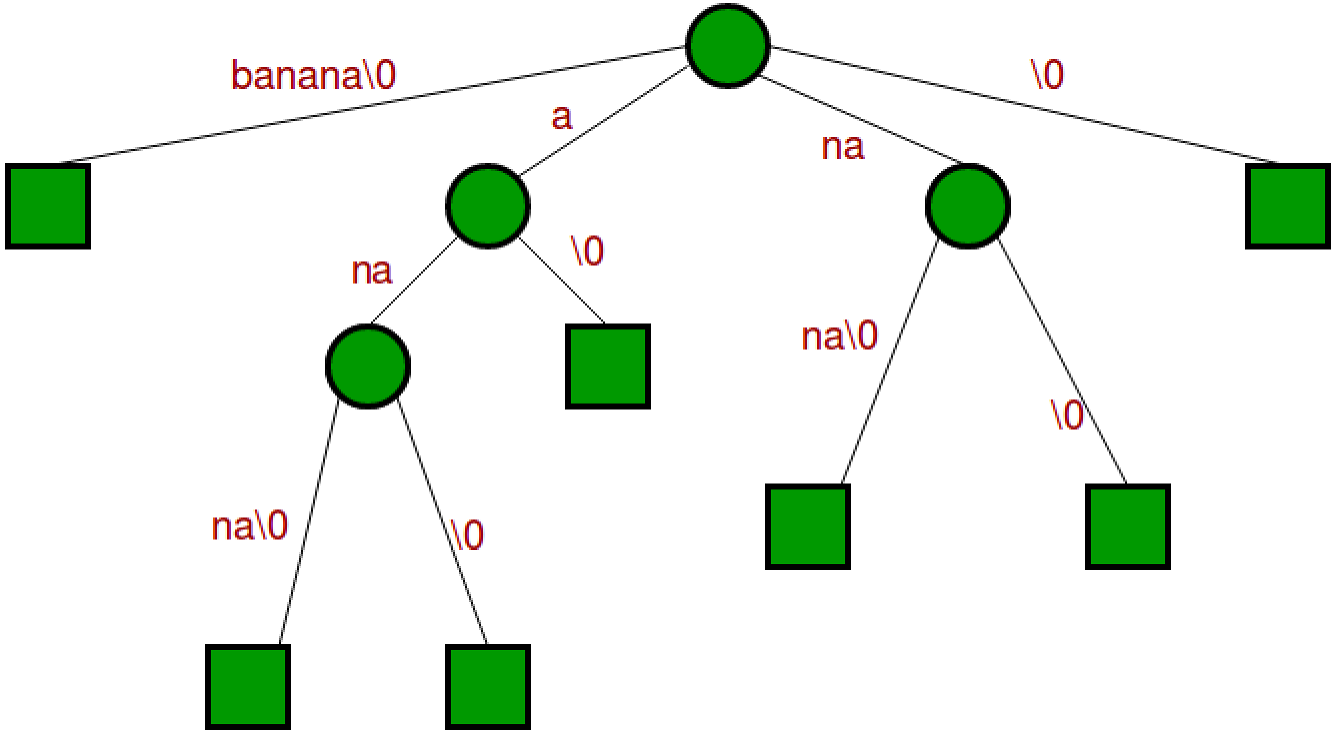
Standard Trie of all Suffixes: “banana\0”



https://www.geeksforgeeks.org/

Suffix Tree

- A Suffix Tree for a given text is a compressed trie for all suffixes of the given text.
- If we join chains of single nodes, we get the following compressed trie, which is the Suffix Tree for given text “banana\0”



THANK YOU!

Next Class: Graph Algorithms…