String: Pattern Matching

What is Pattern Matching?

- Definition:
  - given a text string T and a pattern string P.
  - Find the pattern inside the text and return the starting index where pattern p appears in Text t.
    - T: "IIITDM Jabalpur"
    - P: "bal"
- Applications:
  - text editors, Web search engines (e.g., Google), image analysis

String Concepts

- Assume *S* is a string of size *m*.
- A *substring* *S[i .. j]* of *S* is the string fragment between indexes *i* and *j*.
- A *prefix* of *S* is a substring *S[0 .. i]*
- A *suffix* of *S* is a substring S[i .. m-1]
  - *i* is an index between *0* and *m-1*

Examples

S	a	n	d	r	e	w
	0					5

- Substring S[1..3] == "ndr"

- All possible prefixes of S:
  - "andrew", "andre", "andr", "and", "an”, "a"

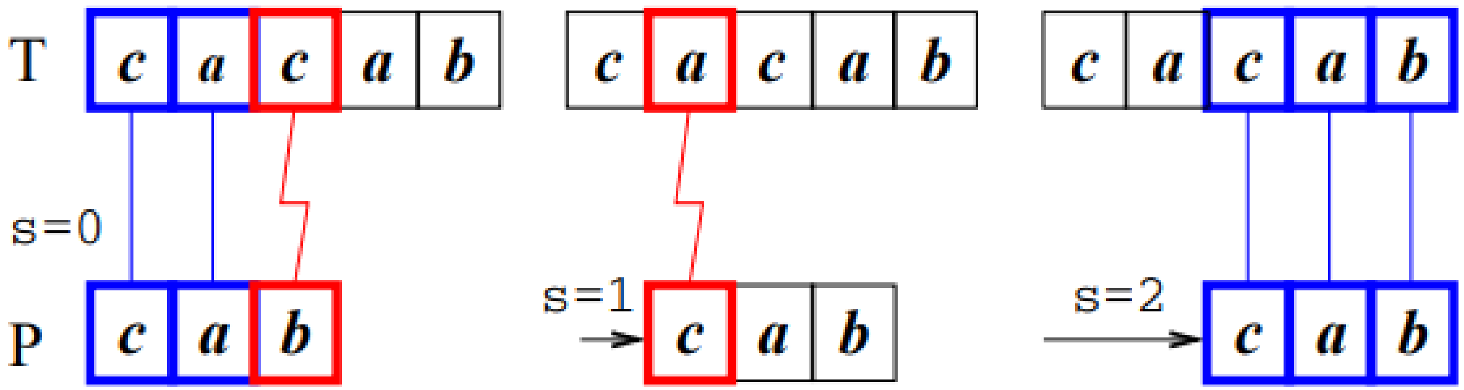
- All possible suffixes of S:
  - "andrew", "ndrew", "drew", "rew", "ew", "w"

The Brute Force Algorithm

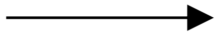
- The Brute-force (Naïve) pattern matching algorithm compares the pattern P with the text T for each possible shift of P relative to T, until either
  - a match is found, or
  - all placements of the pattern have been tried

The Brute Force Algorithm

- Check each position in the text T to see if the pattern P starts at that position.



P moves 1 char at a time through T



. . . .

Brute-force Pattern Matching

```
Algorithm-NAVE_STRING_MATCHING (T, P)
for i←0 to n-m do
   if P[1......m] == T[i+1.....i+m] then
       print "Match Found"
  else
       print "Match not Found"
  end
end
```

Brute Force-Complexity

- Given a pattern M characters in length, and a text N characters in length...
- Worst case: compares pattern to each substring of text of length M. For example, M=5.
- This kind of case can occur for image data.

1) *AAAAA*AAAAAAAAAAAAAAAAAAAAH
AAAAH **5 comparisons made**
2) *AAAAA*AAAAAAAAAAAAAAAAAAAAH
AAAAH **5 comparisons made**
3) *AAAAA*AAAAAAAAAAAAAAAAAAAAH
AAAAH **5 comparisons made**
4) *AAAAA*AAAAAAAAAAAAAAAAAAAAH
AAAAH **5 comparisons made**
5) *AAAAA*AAAAAAAAAAAAAAAAAAAAH
AAAAH **5 comparisons made**

....

N) AAAAAAAAAAAAAAAAAAAAA*AAAAH*
5 comparisons made *AAAAH*

Total number of comparisons: ***M (N-M+1),*** because maximum number of shifting will be equal to ***N-M+1.***

Worst case time complexity: **O(MN)**

Brute Force-Complexity(cont.)

- Given a pattern of M characters in length, and a text of N characters in length...
- Best case if pattern found: Finds pattern in first M positions of text. For example, M=5.

1) *AAAAA*AAAAAAAAAAAAAAAAAAAAAH
AAAAA **5 comparisons made**

Total number of comparisons: M
Best case time complexity: O(M)

Brute Force-Complexity(cont.)

- Given a pattern M characters in length, and a text N characters in length...
- Best case if pattern not found: Always mismatch on first character. For example, M=5.

```
1) AAAAAAAAAAAAAAAAAAAAAAAAAAAAH
   OOOOH        1 comparison made
2) AAAAAAAAAAAAAAAAAAAAAAAAAAAAH
    OOOOH       1 comparison made
3) AAAAAAAAAAAAAAAAAAAAAAAAAAAAH
     OOOOH      1 comparison made
4) AAAAAAAAAAAAAAAAAAAAAAAAAAAAH
      OOOOH     1 comparison made
5) AAAAAAAAAAAAAAAAAAAAAAAAAAAAH
       OOOOH    1 comparison made
   ...
N) AAAAAAAAAAAAAAAAAAAAAAAAAAAAH
        1 comparison made   OOOOH
```

Total number of comparisons: ***M (N-M+1)***

Worst case time complexity: **O(MN)**

Pattern Matching using Finite State Automata

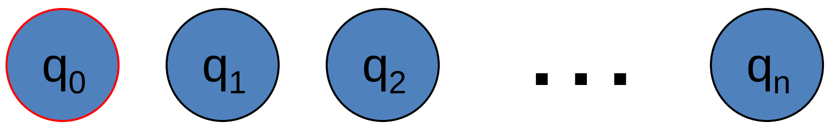
The FSM-based string-matching algorithm is very efficient since it examines each text character at only once.

The time complexity of the FSM method for pattern matching is O(n).

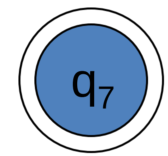
In this approach, FSM is created for the pattern p and then each character of the text T is examined using the FSM matching algorithm whether the pattern appears in the text or not.

Pattern Matching using Finite State Automata

- An FSA is defined by 5 components
  - Q is the set of states

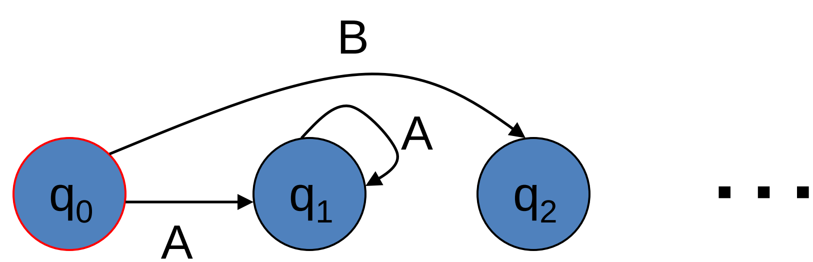


  - q_0 is the start state
  - $A \subseteq Q$, is the set of accepting states where $|A| > 0$

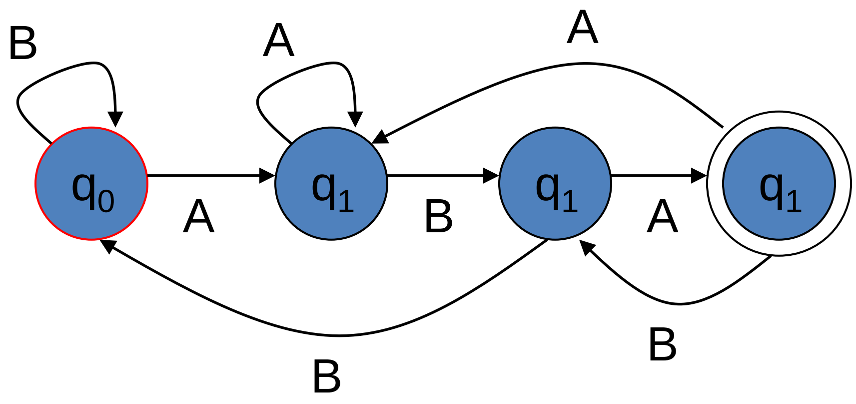


  - Σ is the alphabet e.g. {A, B}
  - δ is the transition function from Q x Σ to Q

Q Σ	Q
q_0 A	q_1
q_0 B	q_2
q_1 A	q_1



Pattern Matching using Finite State Automata



An FSA starts at state q_0 and reads the characters of the input string one at a time.

If the automaton is in state q and reads character a, then it transitions to state $\delta(q, a)$.

If the FSA reaches an accepting state ($q \in A$), then the FSA has found a match.

Building a string-matching automata

- Given a pattern $P = p_1, p_2, \ldots, p_m$, and text $T = t_1, t_2, \ldots t_n$. Find where pattern P appears in text T.

P = ababaca

T= abababacaba

First, construct an FSM for pattern P.

Prefixes and suffixes will be required to construct FSM for a given pattern.

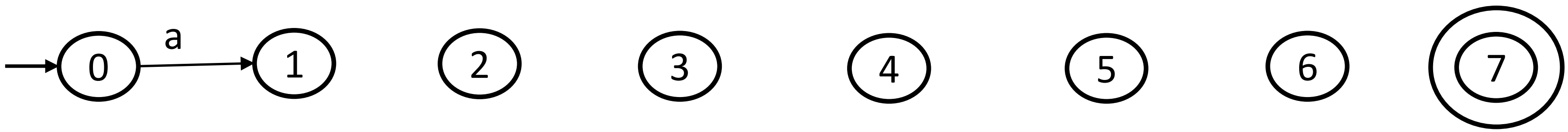
Building a string-matching automata

P = ababaca

T= abababacaba

Since there are 7 symbols in pattern string. So, there will be 8 states in FSM to accept all symbols.

we have three input alphabets *(a, b, c) and* first symbol of pattern is *a*. So, after reading it, it will move to state 2 (state change), no need to check for prefix/ suffix. For other alphabets, there will no move because no prefix/ suffix match *(0b and 0c).*



P = ababaca

Building a string-matching automata

P = ababaca

T= abababacaba

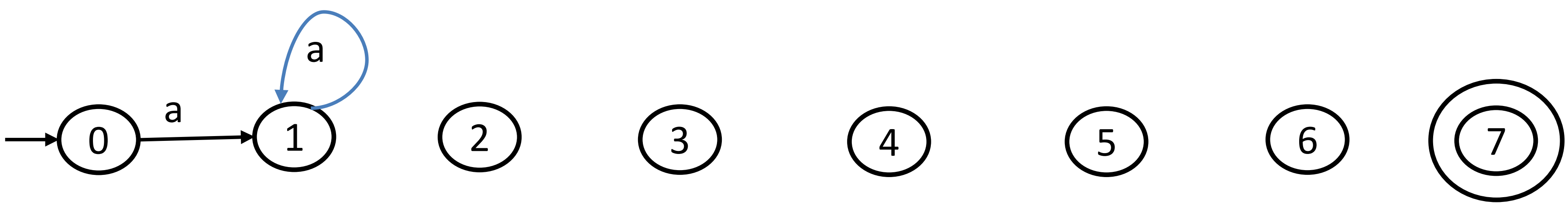
Second symbol is *b*, before reading *b* check for prefixes of all input alphabets ***(a, c)*** for which state changes do not occur.

prefix

Check for input *a* form state *1*, the string from starting state will be *a*|*a.* where *a* and *a* will prefix and suffix.

suffix

The length of prefix/ suffix is *1* . So, there will be self loop at state *1* for input alphabet *a*.



P = ababaca

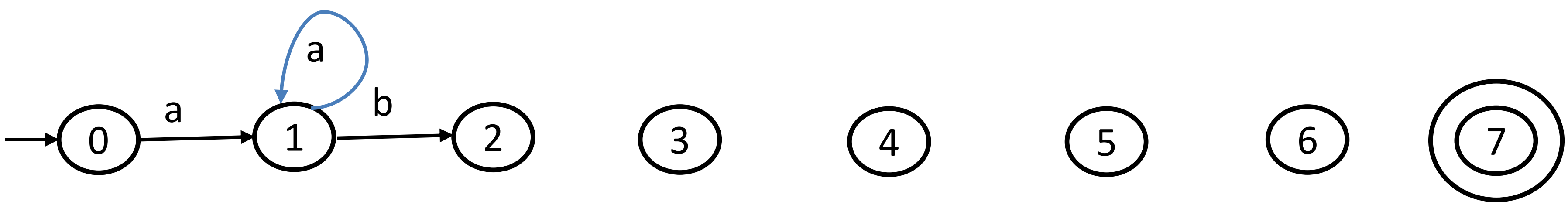
Building a string-matching automata

P = ababaca

T= abababacaba

For *b*, the string will be *ab* and there is a state change, no need to check for prefix/suffix. So, it will move to state 2.

For alphabet *c* prefix/ suffix is *ac*. No prefix/suffix match, so there will be no move for *c*.



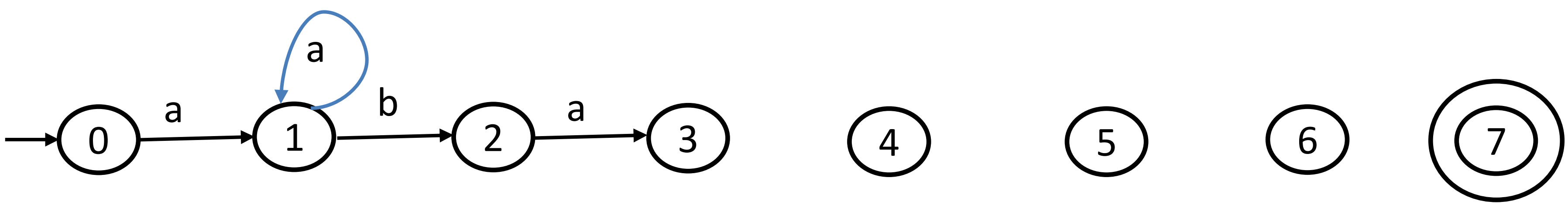
P = ababaca

Building a string-matching automata

P = ababaca

T= abababacaba

Third symbol is *a*, the string will be *aba* and there is a state change, no need to check for prefix/suffix for a. So, it will move to state 3.
Check for symbol *b* and *c.*
For *b* and *c,* string will be *abb* and *abc,* in which no common prefix and suffix. So, there will be no move for *b* and *c*.



P = ababaca

Building a string-matching automata

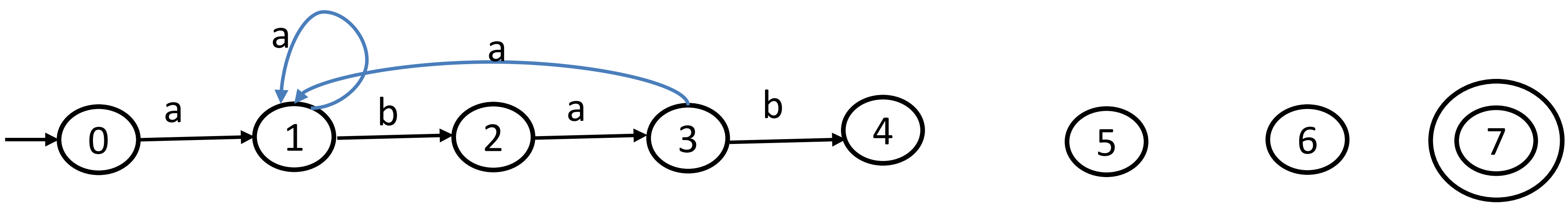
P = ababaca

T= abababacaba

Fourth symbol is *b*, the string will be *abab* and there is a state change, no need to check for prefix/suffix for b. So, it will move to state 4.

Check for symbol *a* and *c.*

For *a,* string will be *abaa,* and there is a common prefix and suffix of length 1 (abaa). So, it will move to state *1* for *a*. For *c,* string will be *abac,* and there is no common prefix/suffix. So, there will no move for *c*.



P = ababaca

Building a string-matching automata

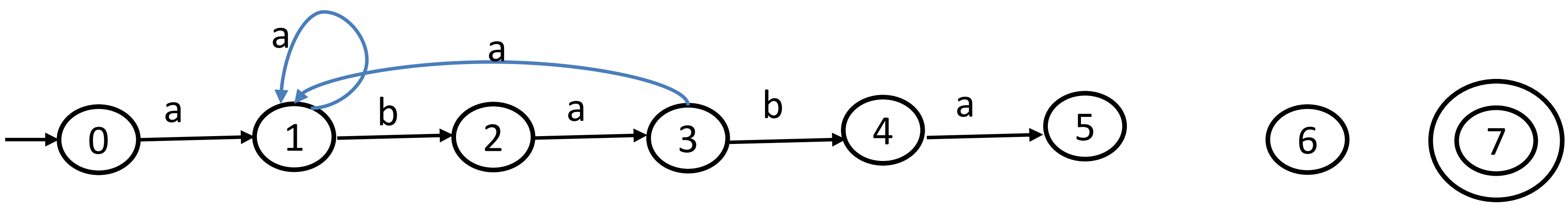
P = ababaca

T= abababacaba

Fifth symbol is *a*, the string will be *ababa* and there is a state change, no need to check for prefix/suffix for a. So, it will move to state 5.

Check for symbol *b* and *c.*

For *b,* string will be *ababb,* and there is no common prefix/ suffix. So, there will be no move for *b*. For *c*, string will be *ababc,* and there is no common prefix/ suffix. So, there will be no move.



P = ababaca

Building a string-matching automata

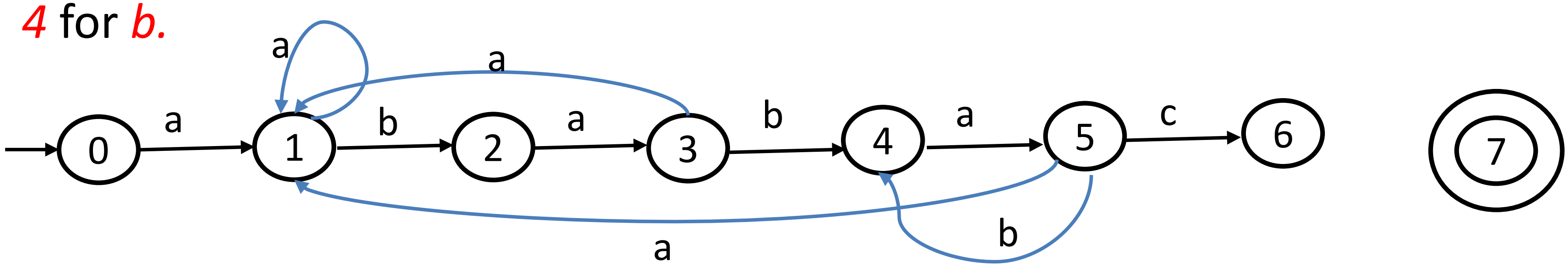
P = ababaca

T= abababacaba

Sixth symbol is *c*, the string will be *ababac* and there is a state change, no need to check for prefix/suffix for c. So, it will move to *state 6*.

Check for symbol *a* and *b.*

For *a,* string will be *ababaa,* and there is a common prefix/ suffix of *length 1 (ababaa)*. So, it will move to state *1* for *a*. For *b,* string will be *ababab,* there is a common prefix/ suffix of *length 4* *(a b a b a b).* So, it will move to state *4* for *b.*



P = ababaca

Building a string-matching automata

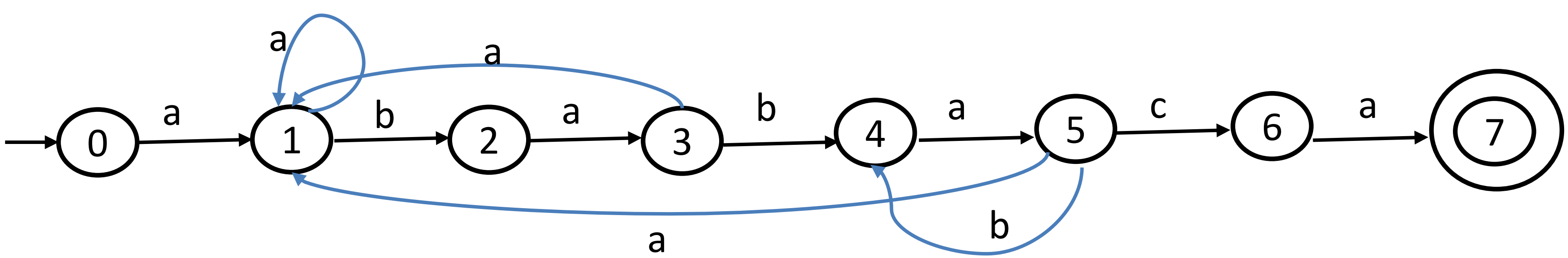
P = ababaca

T= abababacaba

Seventh symbol is *a*, the string will be *ababaca* and there is a state change, no need to check for prefix/suffix for a. So, it will move to *state 7*.

Check for symbol *b* and *c.*

For *b,* string will be *ababacb,* and there is no common prefix/suffix. So, no move for *b*. For *c,* string will be *ababacc,* and there is no common prefix/suffix. So, no move for *c*.



P = ababaca

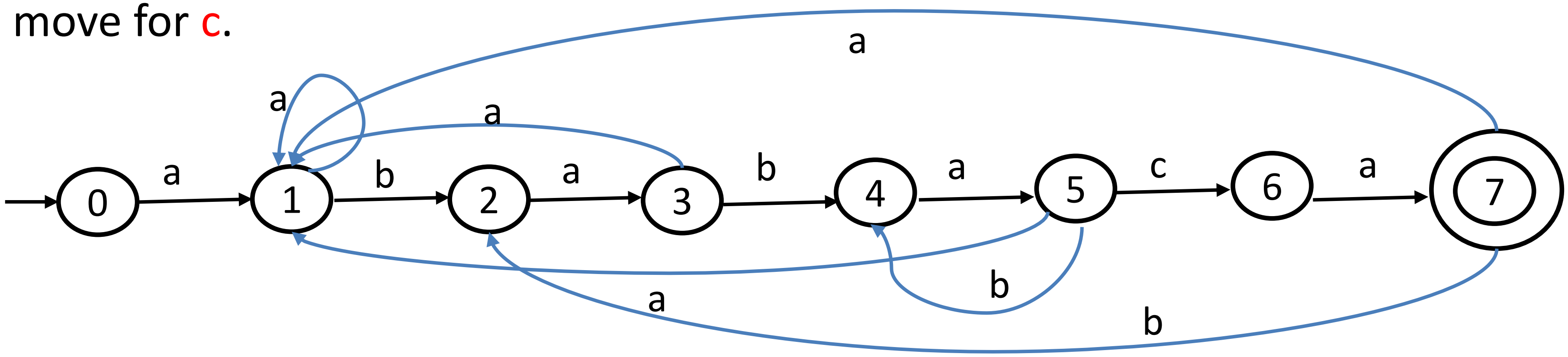
Building a string-matching automata

P = ababaca   T= abababacaba

All input alphabets have been traversed and we reached to last state. So, lets check, if there is any transition possible from final state.

Check for symbol *a, b,* and *c.*

For *a,* string will be *ababacaa,* and there is a common prefix/suffix of length 1. So, there will be a move from *state 7* to *state 1*. For *b*, string will be *ababacab* and there is a common prefix/suffix of length 2. So, there will be a move from *state 7* to *state 2*. For *c*, string will be *ababacac,* there is no common prefix/suffix. So, no move for c.



P = ababaca

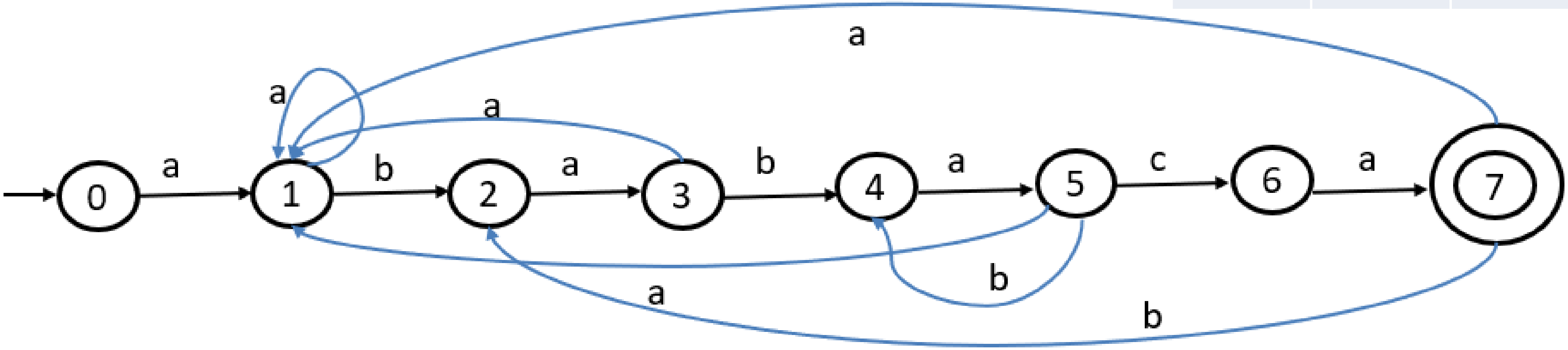
Building a string-matching automata

P = ababaca   T= abababacaba

This is the finite automata for the given pattern. Let's create transition table and use the pattern matching algorithm to check position, where given pattern appears in the text.

Transition table

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



Building a string-matching automata

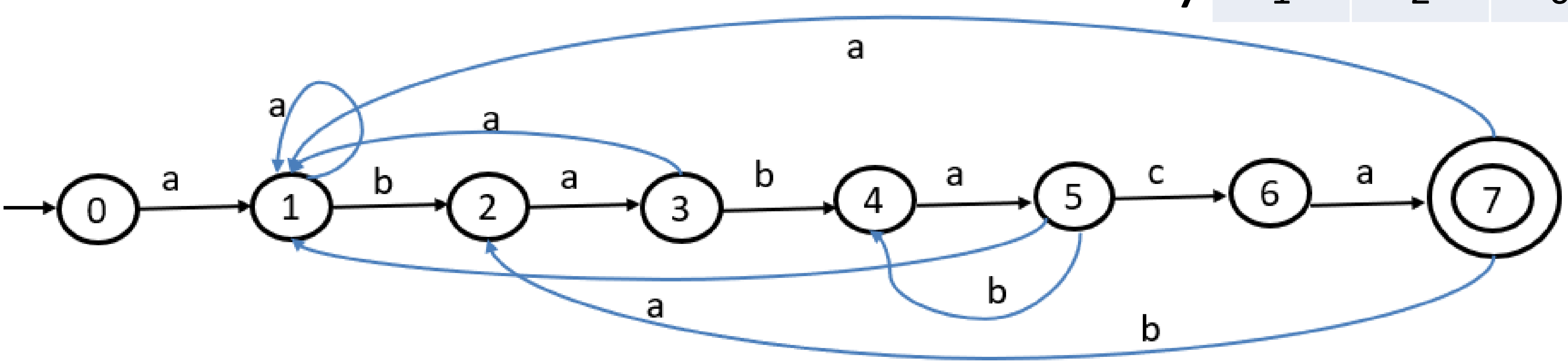
P = ababaca   T= abababacaba

Finite_automata (T, δ, m)

1. n <- length (T)
2. q<- 0
3. for i <- 1 to n
    1. do q <- δ(q, T[i])
    2. if (q==m) then,
        1. Print pattern occur with shift "i-m"

Transition table

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



Building a string-matching automata

P = ababaca   T= a b a b a b a c a b a
1 2 3 4 5 6 7 8 9 10 11

Transition table

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

Finite_automata (T, δ, m)

1. n <- length (T)   **11**
2. q<- 0   **State 0**
3. for i <- 1 to n
   1. do q <- δ(q, T[i])
   2. if (q==m) then,   **m= pattern size =7**
      1. Print pattern occur with shift "i-m"

1. (q0, T[1]) => (q0, a) => q1

2. (q1, T[2]) => (q1, b) => q2, 3. (q2, a) => q3, 4. (q3, b) => q4, 5. (q4, a) =>q5

6. (q5, b) => q4, 7. (q4, a) =>q5, 8. (q5, c) => q6, 9. (q6, a) =>q7,

Here, q=m, (both are 7), then *pattern p* occur in *text t* with a shift of (i-m), 9-7=2 onwards.

T= a b *a b a b a c a* b a
1 2 3 4 5 6 7 8 9 10 11

Building a string-matching automata

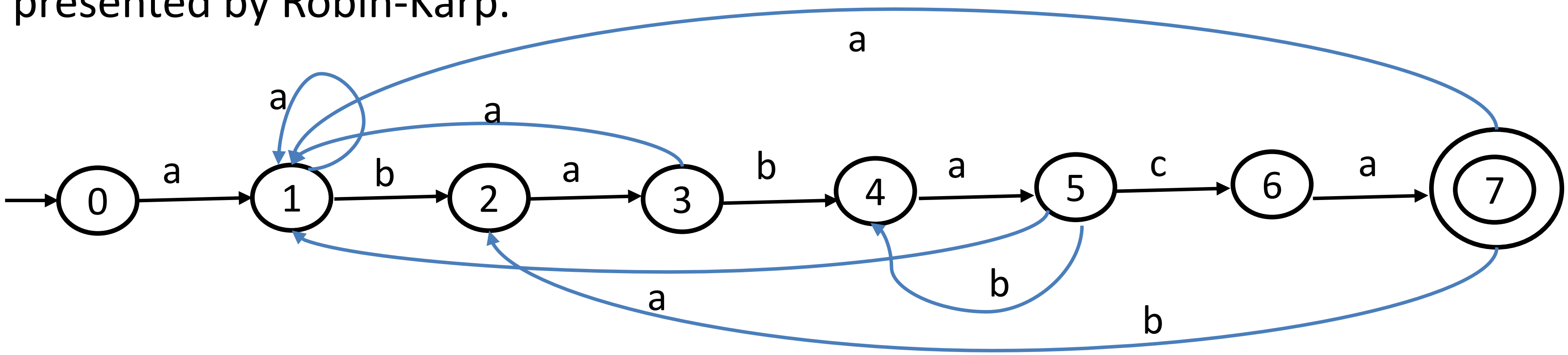
Time complexity:

Once we have *constructed a finite automaton* for the pattern, searching a text for the pattern works wonderfully.

- Search time is *O(n)*.
- Each character in the text is examined just once, in sequential order.

However, construction of finite automata takes $O(m^3\ |\Sigma|)$ time in worst case, where m is the length of pattern and $|\Sigma|$ is number of input symbols.

Therefore, to reduce the search complexity, a new search algorithm has been presented by Robin-Karp.



P = ababaca

Rabin-Karp Algorithm

- The Rabin-Karp string searching algorithm calculates a **hash value** for the *pattern*, and for each *M-character* subsequence of *text and then the hash value of pattern and* the hash value of *M-character* subsequence of *text* are compared.
- If the *hash values* are *unequal*, the algorithm will *calculate* the *hash value* for *next M-character* sequence.
- If the hash values are equal, the algorithm will compare the pattern and the M-character sequence with a Brute Force comparison. If there is a match, it is called a hit; otherwise, it is called a spurious hit.
- In this way, there is *only one* comparison per text subsequence, and Brute Force is only needed when hash values match.
- This algorithm first encode each character to some numerical value and then it uses hash function.

Rabin-Karp Algorithm

Pattern p is M characters long

hash_p=hash value of pattern

hash_t=hash value of first M letters in body of text

do

    if (hash_p == hash_t)

        brute force comparison of pattern and selected section of text

  else

        hash_t= hash value of next section of text, one character over

while (end of text)

Rabin-Karp algorithm

- Use a hash function T that computes a numerical representation of P
- Calculate T for all m symbol sequences of S and compare

In the following example, there are two unique alphabet, so we can encode A as 1 and B as 2.

P = ABB => 122 Encoded form

S = BABABBABABA => 21212212121 Encoded form

Rabin-Karp algorithm

- Use a function T that computes a numerical representation of P
- Calculate T for all m symbol sequences of S and compare

Hash P

P = ABB => 122

T(P)

S = BABABBABABA => 21212212121

Rabin-Karp algorithm

- Use a function T that computes a numerical representation of P
- Calculate T for all m symbol sequences of S and compare

P = 122

Hash m symbol sequences and compare

$$S = \underbrace{212}_{= T(122)}12212121$$

T(122) = 1+2+2 =5

T(212) = 2+1+2 =5

Hash value are same but there is no match *(spurious hit).*

Rabin-Karp algorithm

- Use a function T that computes a numerical representation of P
- Calculate T for all m symbol sequences of S and compare

P = 122

Hash m symbol
sequences and compare

No match

$$S = 2\underbrace{121}_{\substack{= \\ T(122)}}2212121$$

T(122) = 1+2+2 =5

T(121) = 1+2+1 =4

Hash values are not same

Rabin-Karp algorithm

- Use a function T that computes a numerical representation of P
- Calculate T for all m symbol sequences of S and compare

P = 122

Hash m symbol sequences and compare

S = 21212212121

$\underbrace{212}_{=}$

T(122)

T(121) = 1+2+2 =5

T(212) = 2+1+2 =5

Hash value are same but there is no match *(spurious hit)*.

Rabin-Karp algorithm

- Use a function T that computes a numerical representation of P
- Calculate T for all m symbol sequences of S and compare

P = 122

Hash m symbol sequences and compare

S = 21212212121

=

T(122)

T(121) = 1+2+2 =5

T(212) = 1+2+2 =5

Hash value are same and there is match *(hit).*

Rabin-Karp algorithm

- Use a function T that computes a numerical representation of P
- Calculate T for all m symbol sequences of S and compare

P = 122

Hash m symbol
sequences and compare

S = 21212212121

$\ldots \underbrace{121}_{= T(122)}$

T(122) = 1+2+2 =5

T(121) = 1+2+1 =4

Hash value are not same

Rabin-Karp algorithm

The approach may be inefficient and may take *O(mn)* time, if the hash function is not correctly defined.

P = 122

S = 21212212121

$\ldots \underbrace{121}_{=}$

T(122)

Rabin-Karp algorithm

For this approach to be useful/efficient, what needs to be true about T?

P = 122

S = 21212212121

$$\ldots \underbrace{121}_{=\,T(122)}$$

Rabin-Karp algorithm

For this to be useful/efficient, what needs to be true about T?

To improve the efficiency, a hash function must be defined so that number of *collisions/ spurious hit* should be avoided. To attain the same, a new hash function is defined by Robin-karp.

Hash Function

Let *b* be the number of letters in the alphabet. The *first* text *subsequence* of *t[i .. i+m-1]* is mapped to the number t_s using the equation given below.

$$t_s \quad = t[i] * b^{m-1} + T[i+1] * b^{m-2} + \ldots + T[i+m-1]$$

Where m is length of pattern, i=1, and b is number of unique characters in the text string t.

Let's say that alphabet consists of *10* letters i.e., *a, b, c, d, e, f, g, h, i, j* where, *"a"* corresponds to *1*, *"b"* corresponds to *2* and so on, then hash value for string *"cah"* can be computed as follows-

b=10, M=3

$$3*10^{3-1} + 1*10^{3-2} + 8*10^{3-3} = 318$$

The above equation is modified to compute the hash value of pattern P.

Hash Function

The *numeric value* of *first substring* t_0 of *length m* from text *T[1…n]* can be computed in *O(m)* time.

Remaining all t_i, $i = 1, 2, 3, \ldots, n - m$, can be computed in constant time.

Given t_s, we can compute t_{s+1} as,

$$\boxed{t_{s+1} \quad = \quad b(t_s - b^{m-1}\, T[s + 1]) + T[s + m + 1]}$$

Assume that *T = [4, 3, 1, 5, 6, 7, 5, 9, 3]* and *P = [1, 5, 6].* Here length of *P* is *3*, so *m = 3*. Consider that, for given pattern *P*, its value *p = 156*, and $t_1 = 431$.

$$\begin{aligned} t_1 \quad &= \quad 10(431 - 10^2 T[1]) + T[4] \\ &= \quad 10(431 - 400\ ) + 5 \\ &= 315 \end{aligned}$$

The first term is computed using the following formula-

$$\boxed{t_s = \ t[i] * b^{m-1} + T[i+ 1] * b^{m-2} + \ldots . + T[i + m - 1] * \ b^{m-m}}$$

Rabin-Karp algorithm

In some cases, the values of p and t_s may be too large to process. We can reduce these values by taking its modulo with suitable number q. Typically, q is a prime number.

$$\boxed{t_{s+1} = (b*(t_s - T[s + 1]h) + T[s + m + 1]) \bmod q,}$$

Where $h = b^m - 1 \bmod q$

Rabin-Karp Algorithm Complexity

The average case and best-case complexity of Rabin-Karp algorithm is $O(m + n)$ and the worst-case complexity is $O(mn)$.

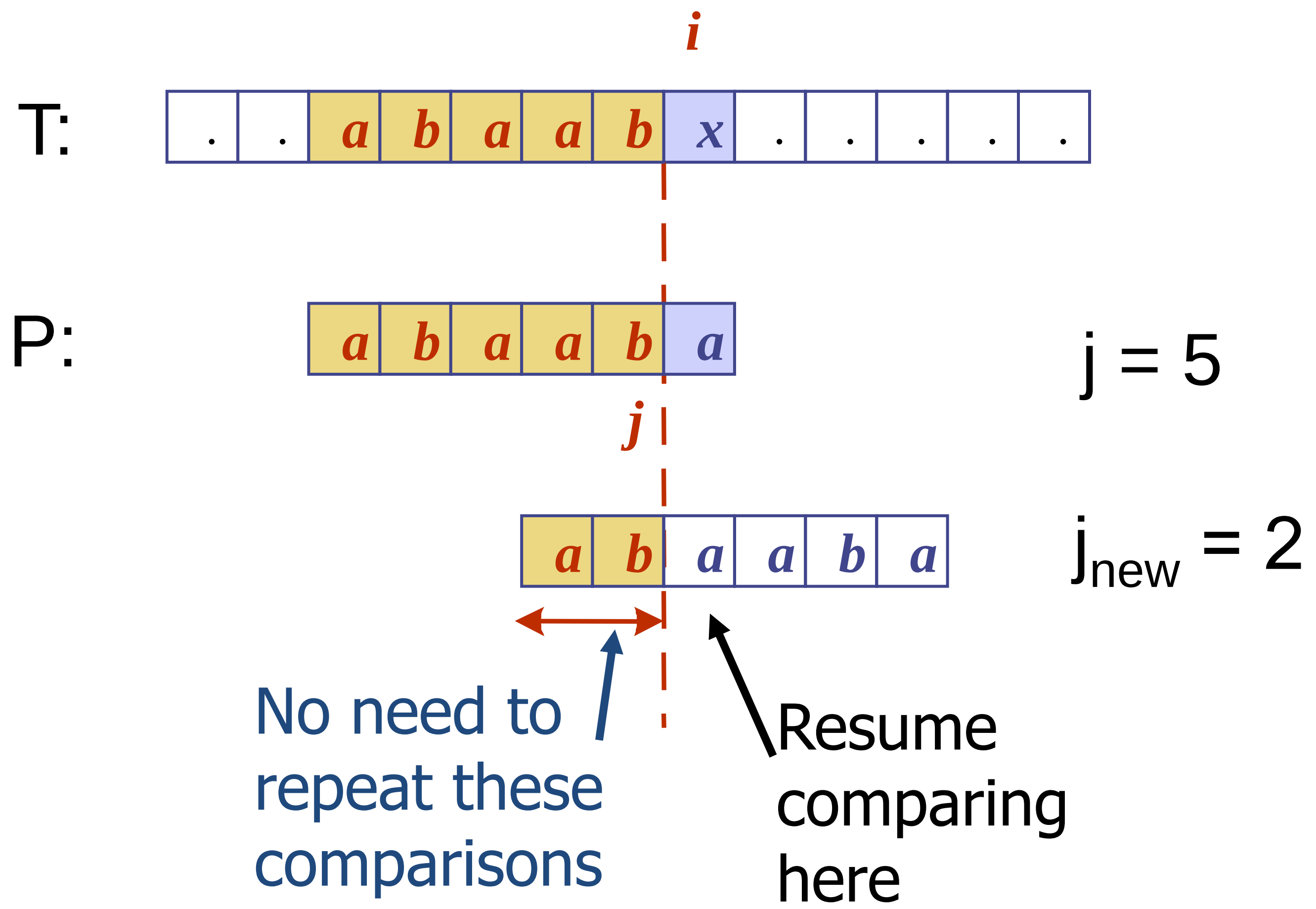
The worst-case complexity occurs when spurious hits occur for all the windows.

The KMP Algorithm

- The Knuth-Morris-Pratt (KMP) algorithm looks for the pattern in the text in a *left-to-right* order (like the brute force algorithm).
- But it shifts the pattern more intelligently than the brute force algorithm.
- If a mismatch occurs between the text and pattern P at P[j], what is the *most,* we can shift the pattern to avoid wasteful comparisons?
  - *Answer*: the largest *prefix* of *P[0 .. j-1]* that is a *suffix* of *P[1 .. j-1]*.
- The prefix function (Π) is used to find the largest prefix that is also suffix of the string.
- This approach is similar to the NFA-to-DFA approach but is implemented more efficiently.

continued

The KMP Algorithm



The prefix function, Π

```
Compute-Prefix-Function (p)
m ← length[p]              //'p' pattern to be matched
Π[1] ← 0
for q ← 2 to m
        k= Π[q-1]
    while k > 0 and p[k+1] != p[q]
        k ← Π[k]
    if p[q] = p[k+1]
        k ← k +1
   Π[q] ← k
return Π
```

Example: compute Π for the pattern ‘p’ below:

p							
	a	b	c	a	b	c	d

Initially: m = length[p] = 7
Π[1] = 0
k = 0

Step 1: q = 2, k=0 and p[2] != p[0+1]
So, Π[2] = k => Π[2] = 0

q	1	2	3	4	5	6	7
p	a	b	c	a	b	c	d
Π	0	0					

Step 2: q = 3, k = 0, and p[3] != p[0+1]
Π[3] = 0

q	1	2	3	4	5	6	7
p	a	b	c	a	b	c	d
Π	0	0	0				

Step 3: q = 4, k = 0, and p[4] = p[0+1]
Π[4] = 0+1=1, k=1

q	1	2	3	4	5	6	7
p	a	b	c	a	b	c	d
Π	0	0	0	1			

Step 4: q = 5, k =1, and p[5] = p[1+1]

Π[5] = 1+1=2

q	1	2	3	4	5	6	7
p	a	b	c	a	b	c	d
Π	0	0	0	1	2		

Step 5: q = 6, k = 2, and p[6] = p[2+1]

Π[6] = 2+1=3

q	1	2	3	4	5	6	7
p	a	b	c	a	b	c	d
Π	0	0	0	1	2	3	

Step 6: q = 7, k = 2, and p[7] != p[3+1]

K=0

q	1	2	3	4	5	6	7
p	a	b	c	a	b	c	d
Π	0	0	0	1	2	3	0

After iterating 6 times, the prefix function computation is complete:

q	1	2	3	4	5	6	7
p	a	b	c	a	b	c	d
Π	0	0	0	1	2	3	0

KMP Prefix Function Π

- Generate the Π Table for the following-
  - a b c d a b c a b f
  - a a a a b a a c d
  - a b a b a b a b c a

KMP Prefix Function Π

- Generate the Π Table for the following-
  - a b c d a b c a b f
  - a a a a b a a c d
  - a b a b a b a b c a

j	1	2	3	4	5	6	7	8	9	10
***P*[*j*]**	***a***	***b***	***c***	***d***	***a***	***b***	***e***	***a***	***b***	***f***
Π	0	0	0	0	1	2	0	1	2	0

j	1	2	3	4	5	6	7	8	9
***P*[*j*]**	***a***	***a***	***a***	***a***	***b***	***a***	***a***	***c***	***d***
Π	0	1	2	3	0	1	2	0	0

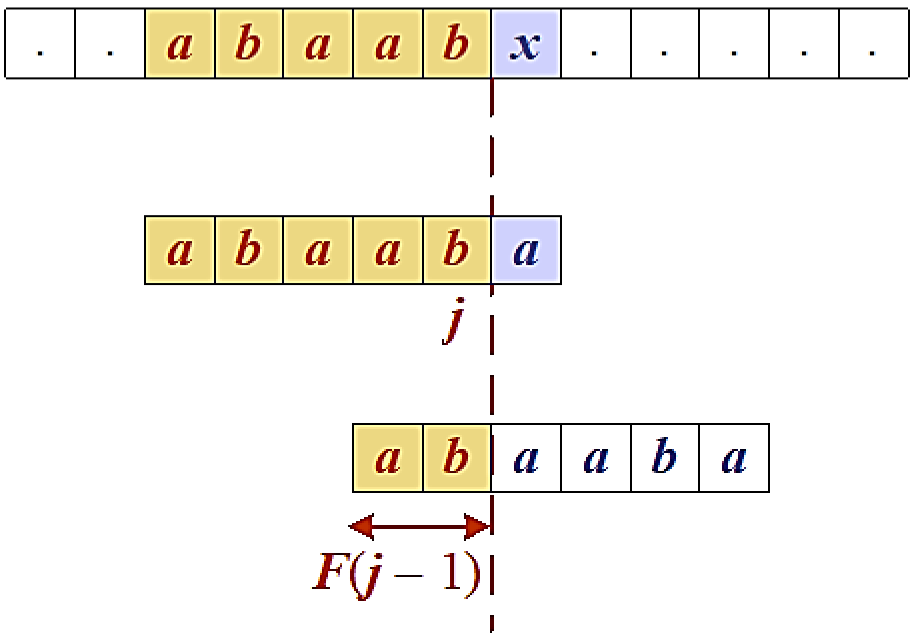
j	1	2	3	4	5	6	7	8	9	10
***P*[*j*]**	***a***	***b***	***a***	***b***	***a***	***b***	***a***	***b***	***c***	***a***
Π	0	0	1	2	3	4	5	6	0	1

KMP Prefix Function Π

- Knuth-Morris-Pratt's algorithm preprocesses the pattern to find matches of prefixes of the pattern with the pattern itself.
- The **failure function** $F(j)$/Π is defined as the length of the longest prefix of $P[0..j]$ that is also a suffix of $P[1..j]$
- Knuth-Morris-Pratt's algorithm modifies the brute-force algorithm so that if a mismatch occurs at $P[j] \neq T[i]$ and $j > 0$, we set $j \leftarrow F(j)$

Π Table

j	1	2	3	4	5	6
$P[j]$	a	b	a	a	b	a
$F(j)$	0	0	1	1	2	1



Prefix Function for Pattern Matching

- Knuth-Morris-Pratt's algorithm modifies the brute-force algorithm.
  - Set j=0 and i=1   // assume indexing starts with 1
  - Compare P[j+1] and T[i], if P[j+1] = T[i] then
    - j=j+1 and i=i+1

  if a mismatch occurs at P[j] (i.e., P[j] != T[i]) and J>0, then check the new value of j from prefix table

  j = F(k);   // obtain the new j

  - else if P[j] != T[i] and J==0
    - Increment I i.e. i=i+1

Use KMP and find the position where pattern p appears in text T.

T= ababcabcabababd, P=ababd

Example

T:

b	a	c	b	a	b	a	b	a	b	a	c	a	c	a

P:

a	b	a	b	a	c	a

Π Table

j	1	2	3	4	5	6	7
$P[j]$	a	b	a	b	a	c	a
$F(j)$/ Π	0	0	1	2	3	0	1

KMP Advantages

- KMP runs in optimal time: O(m+n)
  - very fast
- The algorithm never needs to move backwards in the input text, T
  - this makes the algorithm good for processing very large files that are read in from external devices or through a network stream

KMP Disadvantages

- KMP doesn't work so well as the size of the alphabet increases
  - more chance of a mismatch (more possible mismatches)
  - mismatches tend to occur early in the pattern, but KMP is faster when the mismatches occur later

Other pattern matching algorithms

- The Boyer-Moore Algorithm
- Data structures (Tries, Suffix Tree and compressed tries) for strings

Boyer-Moore algorithm

The Boyer-Moore algorithm for pattern matching a pattern P of length m in a text of length n is based on the following two simple heuristics:

- **Reverse-match heuristic:** Compare P with a subsequence of T moving backwards
- **Bad-character heuristic:** If the character of the text which doesn't match with the current character of the pattern is called the Bad Character. Upon mismatch, we shift the pattern until –
  - The mismatch becomes a match.
  - Pattern P moves past the mismatched character.

Boyer-Moore algorithm

Boyer Moore is a combination of following two approaches.

- Bad Character Approach
- Good Suffix Approach

Both of the above Approach can also be used independently to search a pattern in a text. Here Bad-Match Approach has been used.

Boyer Moore algorithm creates a bad match table to find the optimal shift of the pattern p.

Boyer-Moore algorithm

Bad Character Approach

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

1. The mismatch become a match.

- If the mismatch occur, then we see the Bad-Match table for shifting the pattern.

2. 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.

Boyer-Moore algorithm

Good Suffix Approach

Just like bad character heuristic, a preprocessing table is generated for good suffix Approach.

Let t be substring of text T which is matched with substring of pattern P. Now we shift pattern until :

1. Another occurrence of t in P matched with t in T.
2. A prefix of P, which matches with suffix of t
3. P moves past t.

Boyer-Moore algorithm

Bad Character Approach

This approach creates a Bad-Match Table and uses it for pattern matching.

Steps to find the pattern :

Step 1: Construct the bad-symbol shift table.

Step 2: Align the pattern against the beginning of the text.

Step 3: Repeat the following step until either a matching substring is found or the pattern reaches beyond the last character of the text.

Boyer-Moore algorithm

Construction of Bad Match Table

The bad match table computes the bad match value corresponding to each character of pattern p using the following formula.

if $i<n$

$BMT[i]=$ (Length of string - index - 1)

else

$BMT[i]=$ Length of string

The bad match table for string JABALPUR can be constructed as follows:

0	1	2	3	4	5	6	7
J	A	B	A	L	P	U	R

Boyer-Moore algorithm

Construction of Bad Match Table

if i<n

BMT[i]= (Length of string - index - 1)

Else if (i==n)

BMT[i]= Length of string

The bad match table for string JABALPUR can be constructed as follows:

0	1	2	3	4	5	6	7
J	A	B	A	L	P	U	R

Length of string=8

Letter	J	A	B	L	P	U	R	*
Value	7	~~6~~ 4	5	3	2	1	8	8

* represents any character which is not present in pattern

Boyer-Moore algorithm

Text T= “PDPMIIITDMJABALPUR”

Pattern P= “JABALPUR”

Letter	J	A	B	L	P	U	R	*
Value	7	~~6~~ 4	5	3	2	1	8	8

Bad Match Table

P D P M I I I T D M J A B A L P U R

J A B A L P U R

Mismatch

Check Bad match table’s value for T

As T is not available in table shift pattern by to 8th position rightwards

P D P M I I I T D M J A B A L P U R

J A B A L P U R

Mismatch

Check Bad match table’s value for P

As P is available in table shift pattern by to 2 position rightwards

Boyer-Moore algorithm

Text T= “PDPMIIITDMJABALPUR”

Pattern P= “JABALPUR”

Letter	J	A	B	L	P	U	R	*
Value	7	~~6~~ 4	5	3	2	1	8	8

Bad Match Table

P D P M I I I T D M J A B A L P U R

J A B A L P U R

Mismatch
Check Bad match table’s value for P
As P is available in table shift pattern by to 2 position rightwards

P D P M I I I T D M J A B A L P U R

J A B A L P U R

Match
Pattern P is available from 11th index onwards

Boyer-Moore algorithm

Time complexity

- The Boyer-Moore algorithm has a worst-case time complexity of O(nm).
- The worst case is occurring, when all characters of the text and pattern are same. For example, if the text is "SSSSSSSSSSSSS" and the pattern is "SSSSSS".
- However, it can perform much better than that. In fact, in some cases, it can achieve a sublinear time complexity of O(n/m), which means that it can skip some characters in the text without comparing them. This happens when the pattern has no repeated characters or when it has a large alphabet size.

Trie

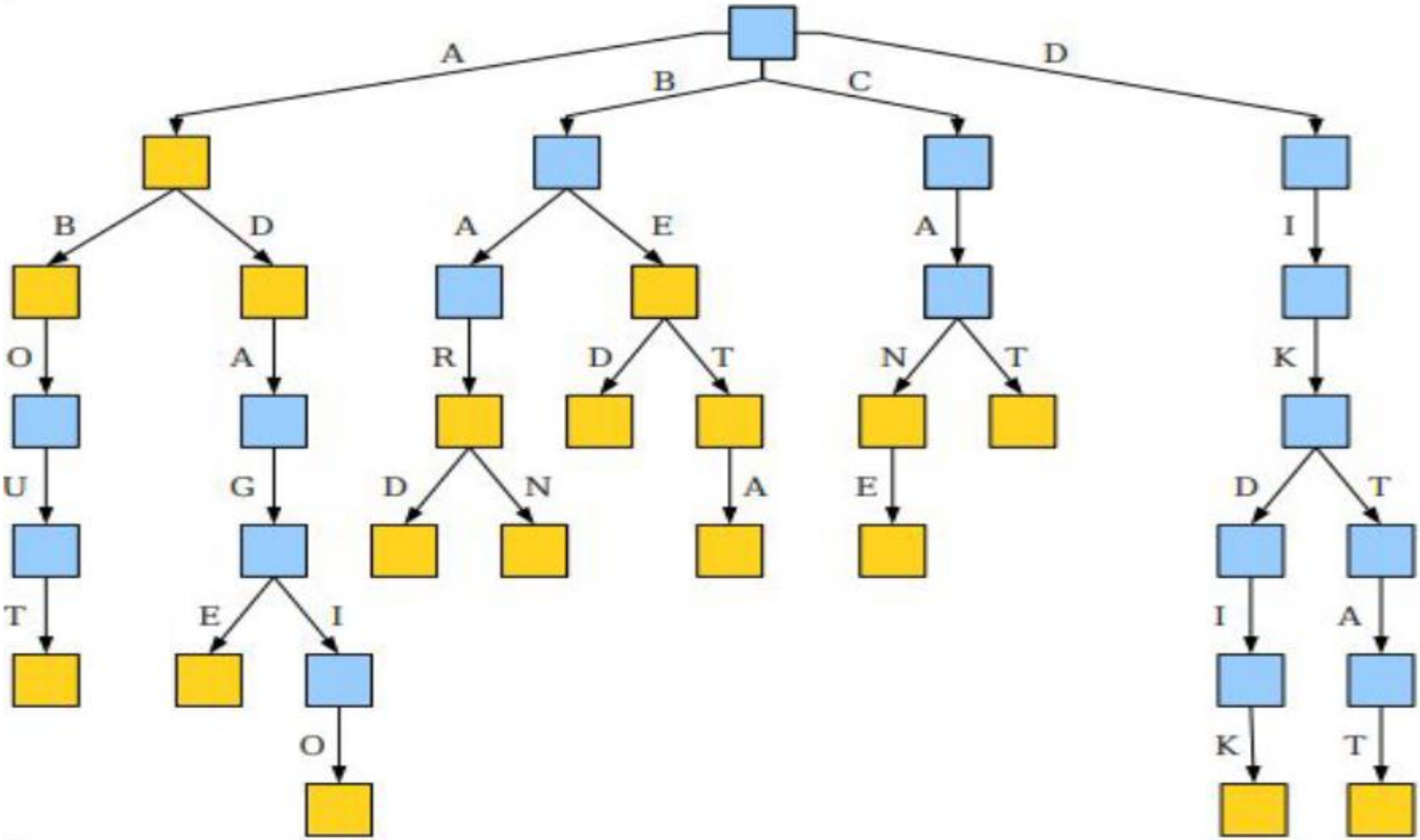
- Tries is an efficient information re***Trie***val data structure.
- Tries can reduce search complexities to optimal limit (key length).
- Binary search tree can reduce retrieval time to **M * log N**, where M is maximum string length and N is number of keys in tree.
- Using Tries, we can search the key in O(M) time, but space complexity for tries can be its limitation.
- All the descendant node in tries have the **same prefix**, hence tries also know as **prefix trees**.

Trie

```
// Trie node
struct TrieNode
{
     struct TrieNode *children[ALPHABET_SIZE];
     // isEndOfWord is true if the node
     // represents end of a word
     bool isEndOfWord;
};
```

- Every node of Trie consists of multiple branches.
- Each branch represents a possible character of keys.
- Last node of every key is marked as end of word node.

Trie Example



Insertion in Trie

- Every character of input key is inserted as an individual Trie node.
- *children* is an array of pointers/references to next level trie nodes.
- The key character acts as an index into the array *children*
- If the input key is new or an extension of existing key, construct non-existing nodes of the key, and mark end of word for last node.
- If the input key is prefix of existing key in Trie, we simply mark the last node of key as end of word.
- The key length determines Trie depth.

Insertion in Trie

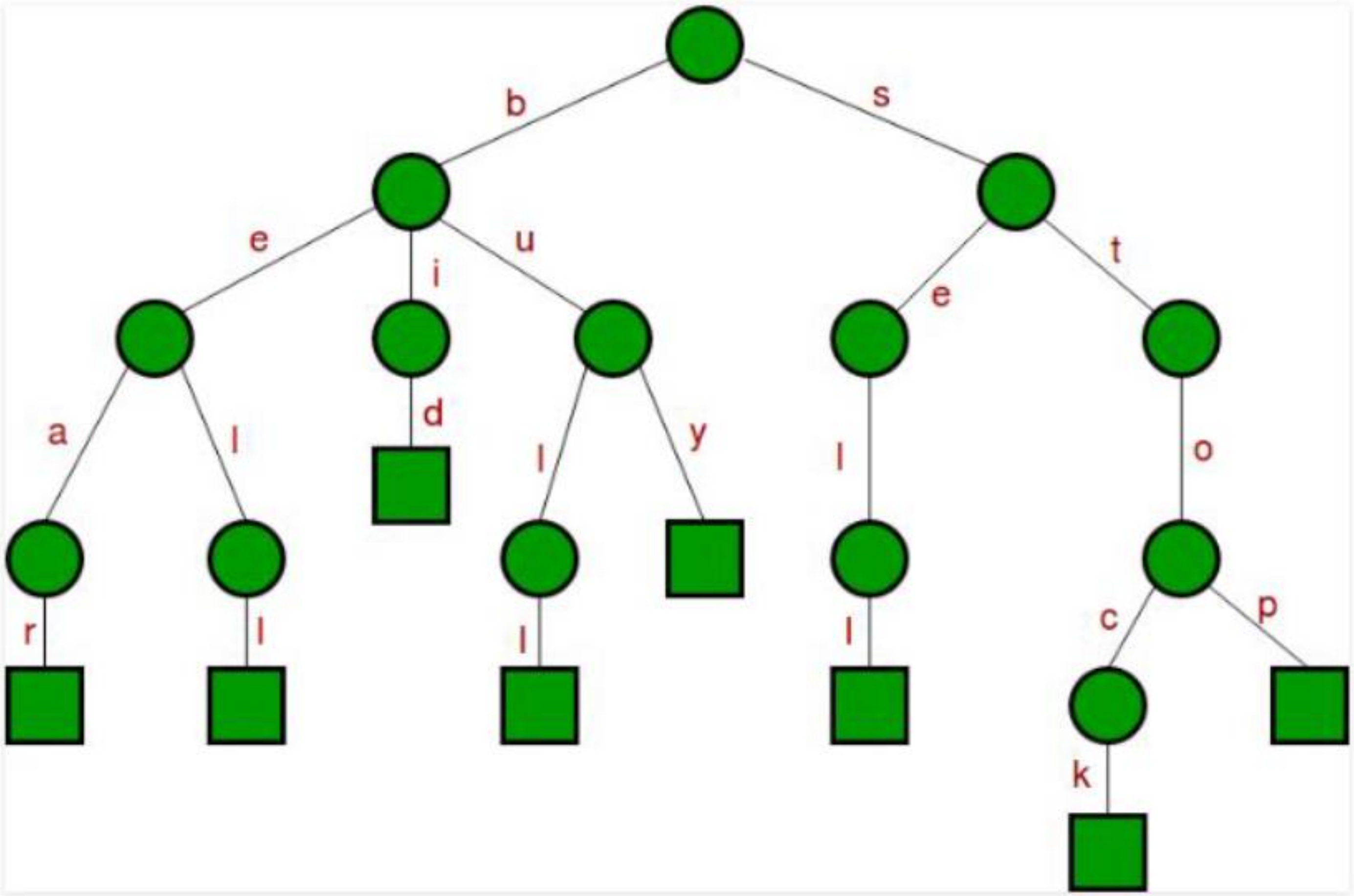
```
         root
      /   \     \
      t   a      b
      |   |      |
      h   n      y
      |   |  \   |
      e   s   y  e
    / |   |
   i  r   w
   |  |   |
   r  e   e
          |
          r
```

void insert(String s) {
for(every char in string s) { if(child node belonging to current char is null) { child node=new Node(); }
current_node=child_node; }

Mark isEndofWord }

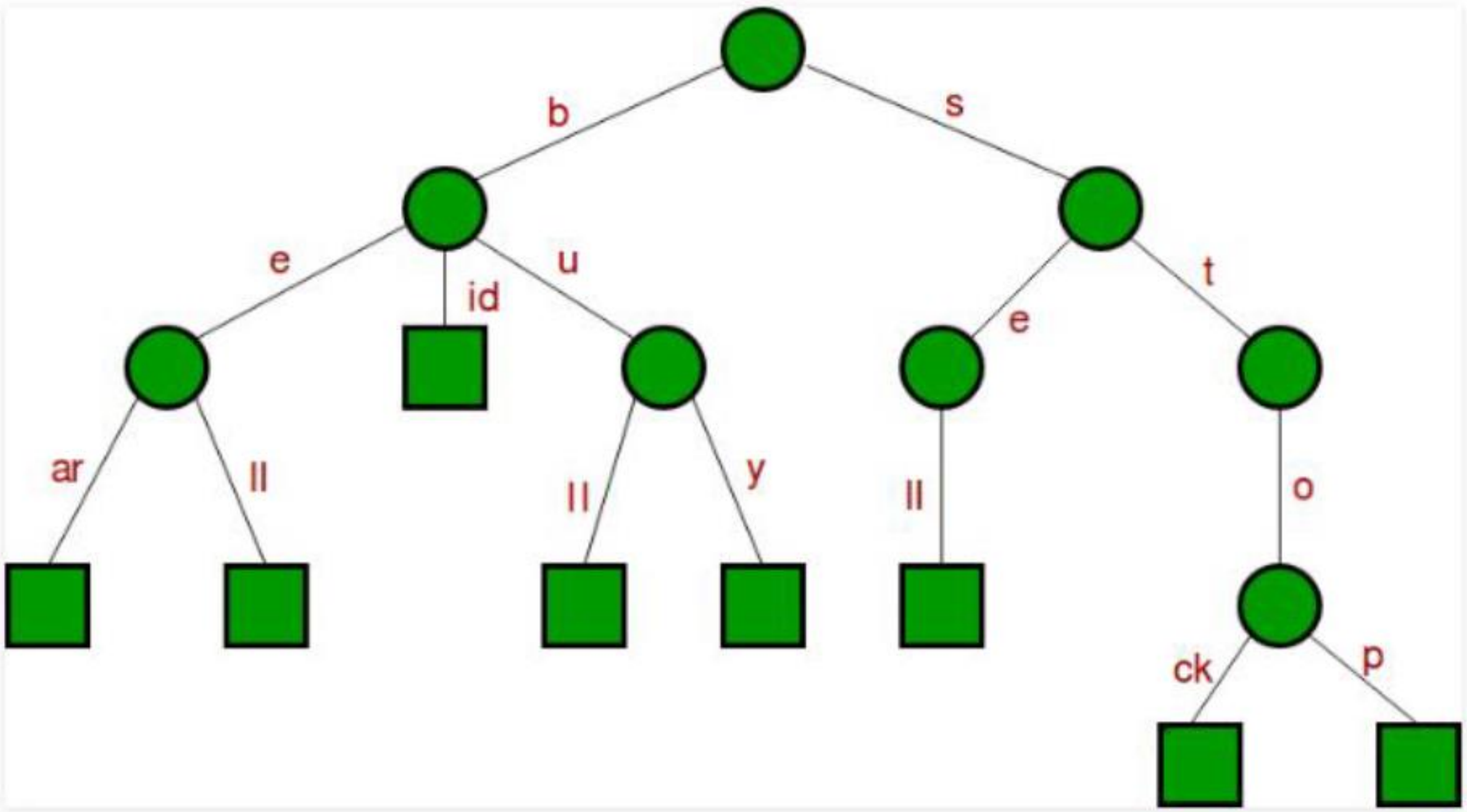
Insertion in Trie

Standard trie- {bear, bell, bid, bull, buy, sell, stock, stop}



Insertion in Trie

Compress Trie - obtained from standard trie by joining chains of single nodes.



Searching in Trie

- Searching for a key is similar to insert operation
- compare the characters and move down.
- search can terminate due to end of string or lack of key in trie.
- In the former case, if the *isEndofWord* field of last node is true, then the key exists in trie.
- In the second case, the search terminates without examining all the characters of key, since the key is not present in trie.

Suffix Trees and Suffix Arrays

Some problems

- Given a pattern P = P[1..m], find all occurrences of P in a text S = S[1..n]
- Another problem:
  - Given two strings $S_1[1..n_1]$ and $S_2[1..n_2]$ find their longest common substring.
    - find i, j, k such that $S_1[i .. i+k-1] = S_2[j .. j+k-1]$ and k is as large as possible.
- Any solutions? How do you solve these problems (efficiently)?

Exact string matching

- Finding the pattern P[1..m] in S[1..n] can be solved simply with a scan of the string S in O(m+n) time. However, when S is very long and we want to perform many queries, it would be desirable to have a search algorithm that could take O(m) time.
- To do that we have to preprocess *S*. The preprocessing step is especially useful in scenarios where the text is relatively constant over time (e.g., a genome), and when search is needed for many different patterns.

Suffix trees

- Any string of length m can be degenerated into m suffixes.
  - abcdefgh (length: 8)
  - 8 suffixes:
    - h, gh, fgh, efgh, defgh, cdefgh, bcefgh, abcdefgh
- The suffixes can be stored in a suffix-tree and this tree can be generated in O(*n*) time
- A string pattern of length *m* can be searched in this suffix tree in O(*m*) time.
  - Whereas, a regular sequential search would take O(*n*) time.

History of suffix trees

- Weiner, 1973: suffix trees introduced, linear-time construction algorithm
- McCreight, 1976: reduced space-complexity
- Ukkonen, 1995: new algorithm, easier to describe
- In this course, we will only cover a naive (quadratic-time) construction.

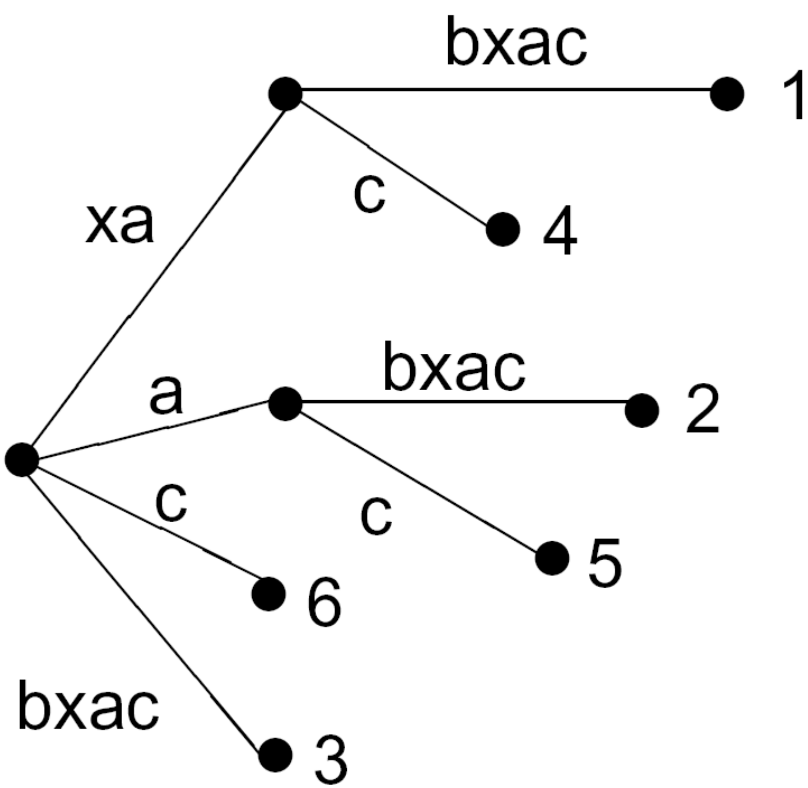
Definition of a suffix tree

- Let $S=S[1..n]$ be a string of length n over a fixed alphabet Σ. A suffix tree for S is a tree with n leaves (representing n suffixes) and the following properties:
  - Every internal node other than the root has at least 2 children
  - Every edge is labeled with a nonempty substring of S.
  - The edges leaving a given node have labels starting with different letters.
  - The concatenation of the labels of the path from the root to leaf i spells out the i-th suffix $S[i..n]$ of S. We denote S[i..n] by S_i.

An example suffix tree

- The suffix tree for string: 1 2 3 4 5 6

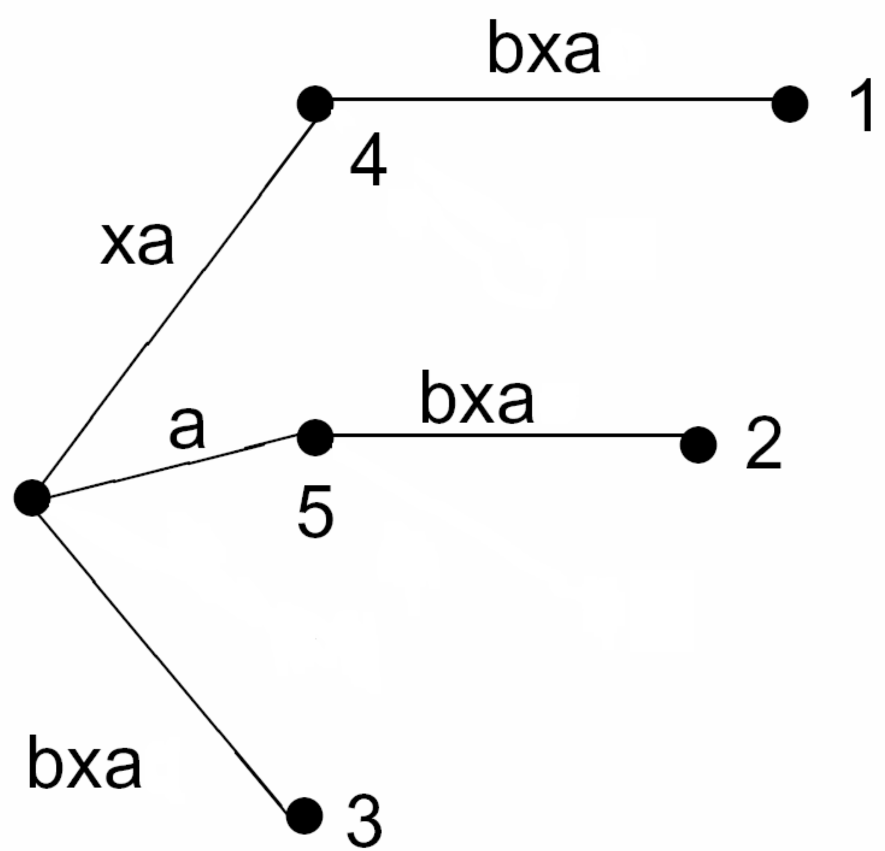
  x a b x a c



Does a suffix tree always exist?

What about the tree for xabxa?

- The suffix tree for string: 1 2 3 4 5
  x a b x a



xa an *a* are not leaf nodes.

Problem

- Note that if a suffix is a prefix of another suffix we cannot have a tree with the properties defined in the previous slides.
  - e.g. *xabxa*

The fourth suffix *xa* or the fifth suffix *a* won’t be represented by a leaf node.

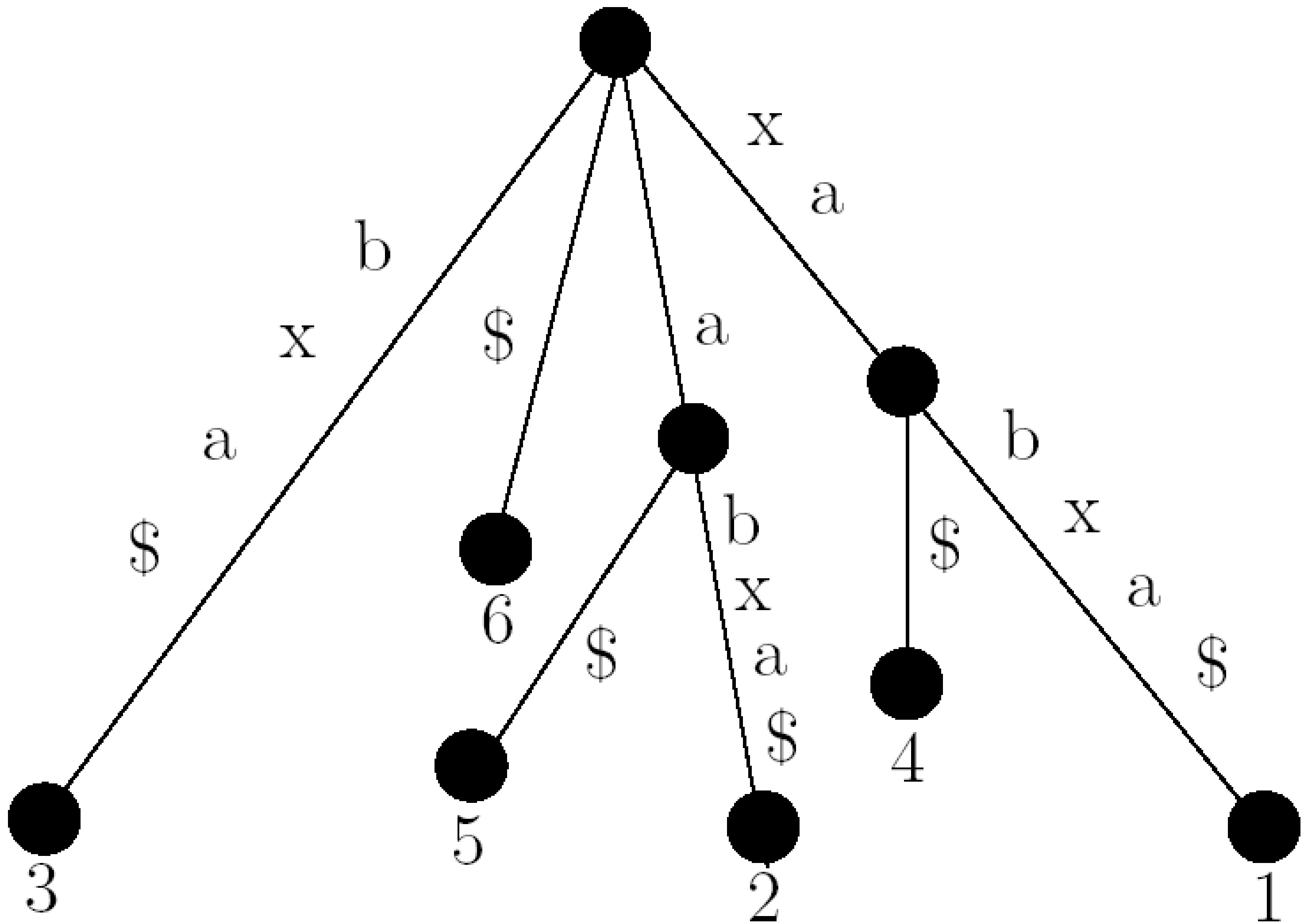
Solution: the terminal character $

- Note that if a suffix is a prefix of another suffix we cannot have a tree with the properties defined in the previous slides.
  - e.g. *xabxa*

The fourth suffix *xa* or the fifth suffix *a* won't be represented by a leaf node.

- Solution: insert a special terminal character at the end such as $. Therefore xa$ will not be a prefix of the suffix xabxa.

The suffix tree for xabxa$



Suffix tree construction

- Start with a root and a leaf numbered 1, connected by an edge labeled *S*$.
- Enter suffixes *S*[2..*n*]$; *S*[3...*n*]$; ... ; *S*[*n*]$ into the tree as follows:
- To insert K_i = *S*[*i*..*n*]$, follow the path from the root matching characters of K_i until the first mismatch at character $K_i[\,j\,]$ (which is bound to happen)

  (a) If the matching cannot continue from a node, denote that node by *w*

  (b) Otherwise the mismatch occurs at the middle of an edge, which has to be split

Suffix tree construction - 2

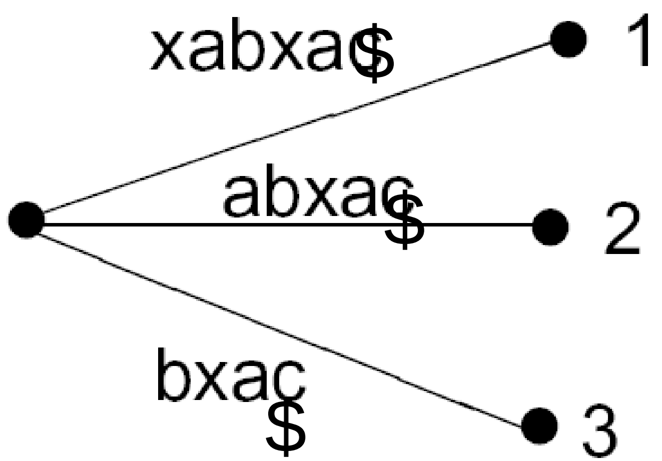
- If the mismatch occurs at the middle of an edge $e = S[u \ldots v]$
  - let the label of that edge be $a_1 \ldots a_l$
  - If the mismatch occurred at character a_k, then create a new node *w*, and replace *e* by two edges S[u ... u+k-1] and S[u+k ... v] labeled by $a_1 \ldots a_k$ *and* $a_{k+1} \ldots a_l$
- Finally, in both cases (a) and (b), create a new leaf numbered *i*, and connect *w* to it by an edge labeled with $K_i[j \ldots |K_i|]$

Example construction

- Let’s construct a suffix tree for xabxac$

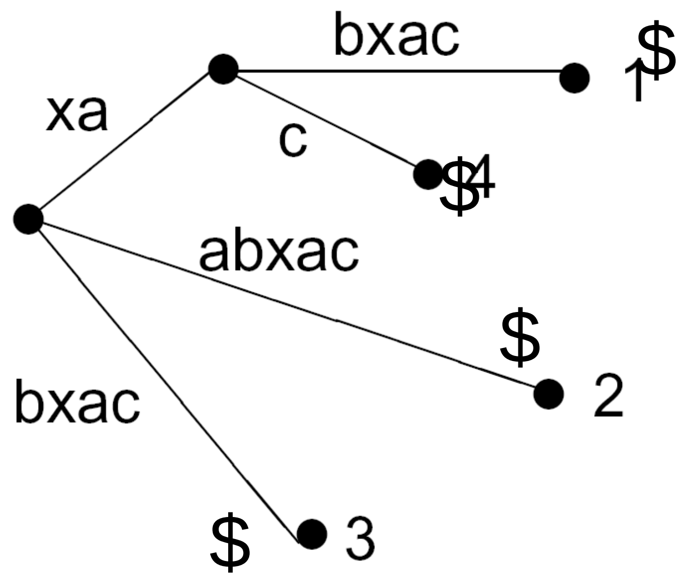
- Start with:

- After inserting the second and third suffix:



Example contd...

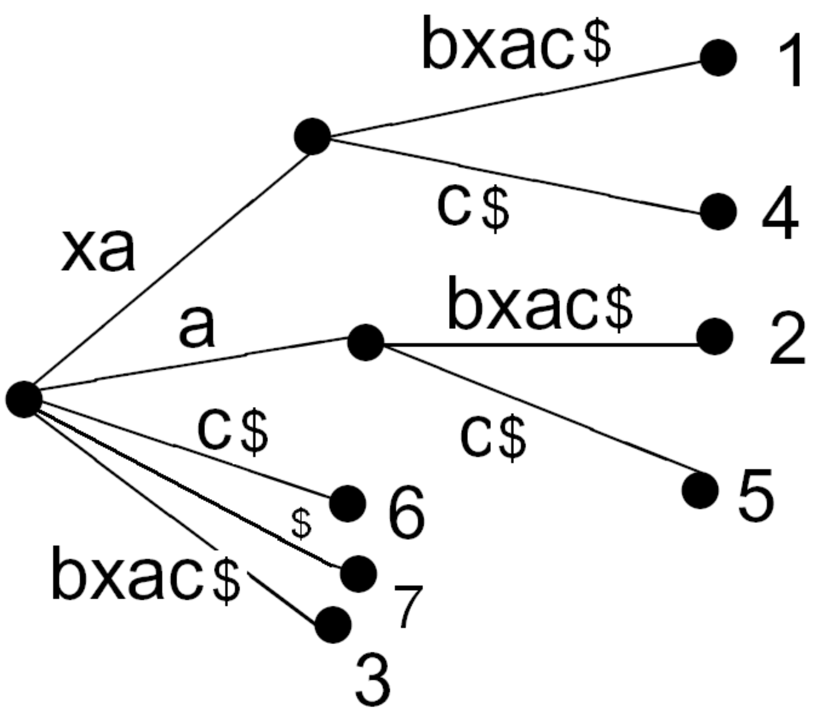
- Inserting the fourth suffix xac$ will cause the first edge to be split:



- Same thing happens for the second edge when ac$ is inserted.

Example contd...

- After inserting the remaining suffixes the tree will be completed:



Complexity of the naive construction

- We need O(n-i+1) time for the i^{th} suffix. Therefore the total running time is:

$$\sum_{1}^{n} O(i) = O(n^2)$$

- What about space complexity?
  - Can also take O(n^2) because we may need to store every suffix in the tree separately,
  - e.g., abcdefghijklmn

Storing the edge labels efficiently

- Note that, we do not store the actual substrings $S[i \ldots j]$ of S in the edges, but only their start and end indices (i, j).
- Nevertheless we keep thinking of the edge labels as substrings of S.
- This will reduce the space complexity to $O(n)$

Using suffix trees for pattern matching

- Given *S* and *P*. How do we find all occurrences of *P* in *S*?
- **Observation.** Each occurrence has to be a prefix of some suffix. Each such prefix corresponds to a path starting at the root.

1. Of course, as a first step, we construct the suffix tree for *S*. Using the naive method this takes quadratic time, but *linear-time* algorithms (e.g., Ukkonen's algorithm) exist.
2. Try to match *P* on a path, starting from the root. Three cases:

   (a) The pattern does not match → *P* does not occur in T

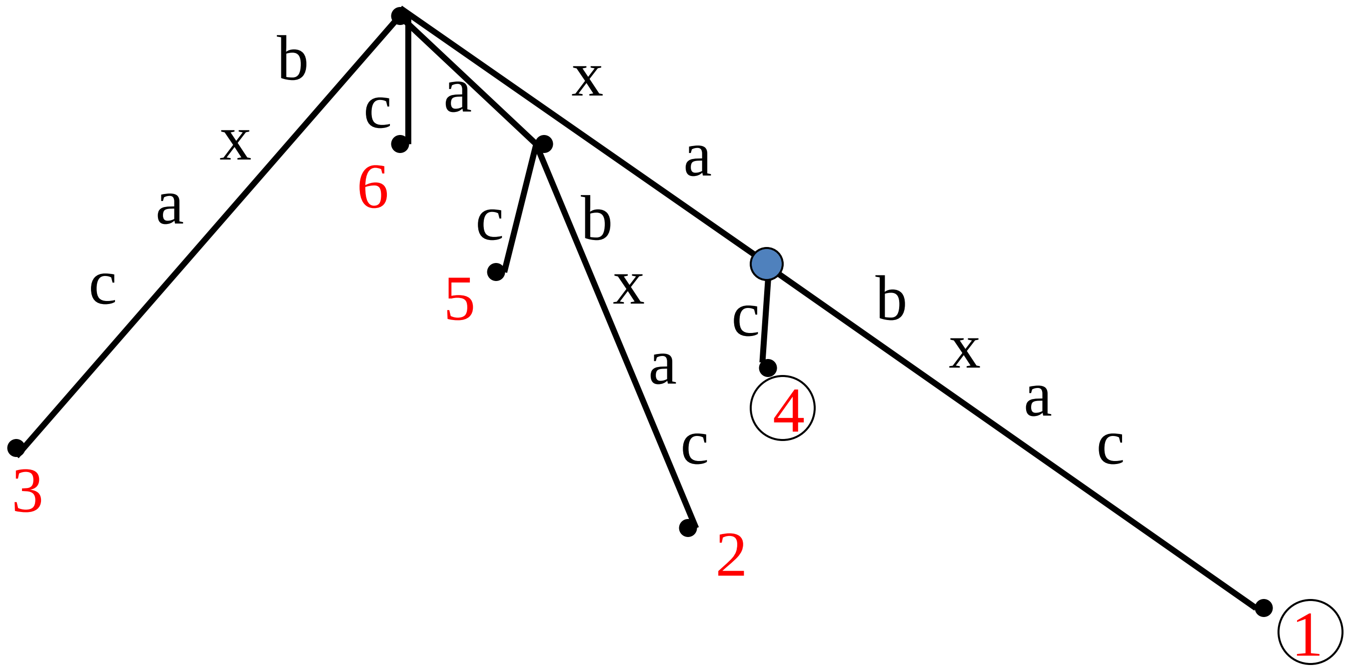
   (b) The match ends in a node *u* of the tree. Set $x = u$.

   (c) The match ends inside an edge (v,w) of the tree. Set $x = w$.

3. All leaves below *x* represent occurrences of *P*.

Illustration

- T = xabxac
  - suffixes ={xabxac, abxac, bxac, xac, ac, c}
- Pattern P_1: xa
- Pattern P_2: xb



Running Time Analysis

- Search time:
  - O(m+k) where k is the number of occurrences of P in T and m is the length of P
  - O(m) to find match point if it exists
  - O(k) to find all leaves below match point

Scalability

- For very large problems a linear time and space bound is not good enough. This lead to the development of structures such as Suffix Arrays to conserve memory .

Two implementation issues

- Alphabet size
- Generalizing to multiple strings

Effects of alphabet size on suffix trees

- We have generally been assuming that the trees are built in such a way that
  - from any node, we can find an edge in constant time for any specific character in Σ
    - an array of size $|\Sigma|$ at each node
- This takes $\Theta(m|\Sigma|)$ space.