String Matching

String Matching Problem

Pattern : compress

Text : We introduce a general framework which is suitable to capture an essence of compressed pattern matching according to various dictionary based compressions. The goal is to find all occurrences of a pattern in a text without decompression, which is one of the most active topics in string matching. Our framework includes such compression methods as Lempel-Ziv family, (LZ77, LZSS, LZ78, LZW), byte-pair encoding, and the static dictionary based method. Technically, our pattern matching algorithm extremely extends that for LZW compressed text presented by Amir, Benson and Farach.

Notation & Terminology

- String S:
  - S[1…n]
- Sub-string of S :
  - S[i…j]
- Prefix of S:
  - S[1…i]
- Suffix of S:
  - S[i…n]
- |S| = n (string length)
- Ex: AGATCGATGGA

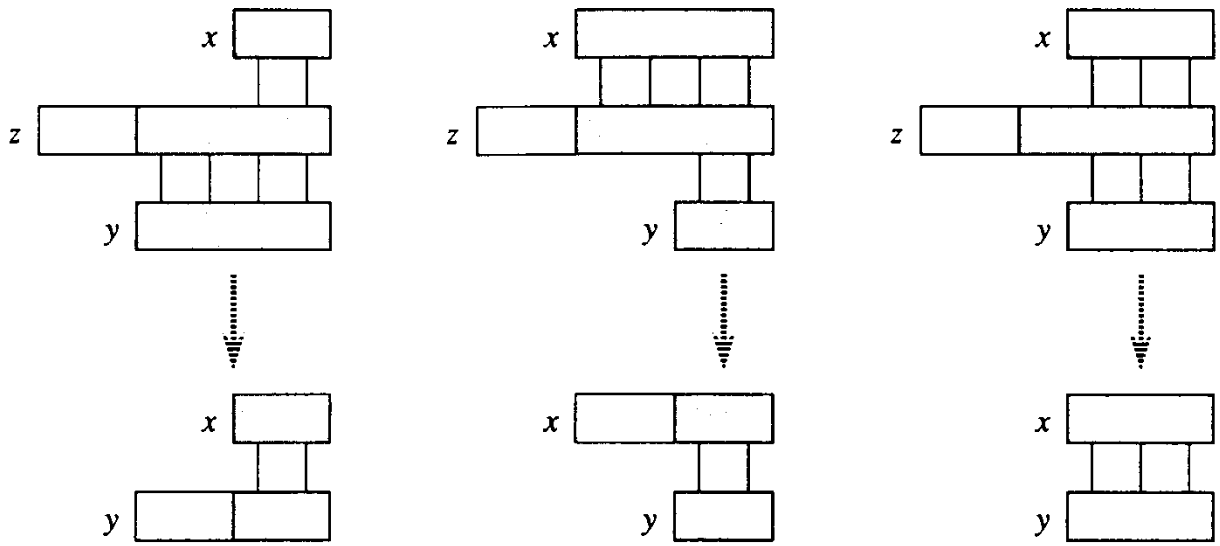
Prefix Sub-string Suffix

Notation

- Σ^* is the set of all finite strings over Σ; length is $|x|$; concatenation of x and y is xy with $|xy| = |x| + |y|$
- w is a prefix of x ($w \sqsubset x$) provided x = wy for some y
- w is a suffix of x ($w \sqsupset x$) provided x = yw for some y

Lemma 34.1 (Overlapping-suffix lemma)

Suppose that x, y, and z are strings such that $x \sqsupset z$ and $y \sqsupset z$. If $|x| \le |y|$, then $x \sqsupset y$. If $|x| \ge |y|$, then $y \sqsupset x$. If $|x| = |y|$, then $x = y$.

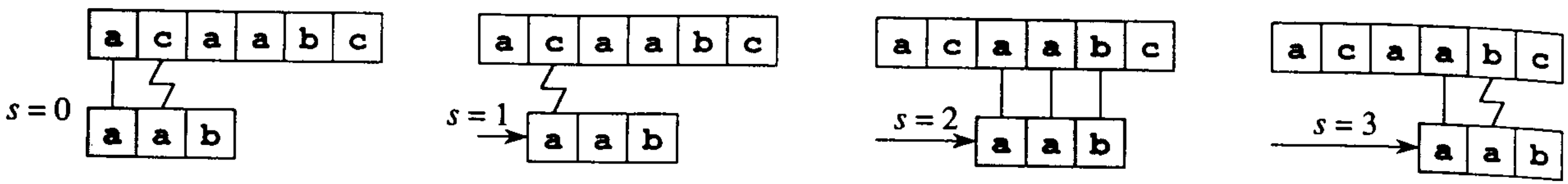


(a) (b) (c)

Naïve String Matcher

```
NAIVE-STRING-MATCHER(T, P)
n ← length[T]
m ← length[P]
for s ← 0 to n − m
        do if P[1 . . m] = T[s + 1 . . s + m]
              then print "Pattern occurs with shift" s
```

- The equality test takes time O(m)



(a) (b) (c) (d)

- Complexity
  - the overall complexity is O((n-m+1)m)
  - if m = n/2 then it is an O(n^2) algorithm

Another method?

- Can we do better than this?
  - Idea: shifting at a mismatch far enough for efficiency not too far for correctness.
  - Ex:

T = x a b x y a b x y a b x z P = a b x y a b x z

a b x y a b x z
X
a b x y a b x z
v v v v v v v X

Jump

a b x y a b x z
v v v v v v v v

➢ Preprocessing on P or T

Two types of algorithms for String Matching

- Preprocessing on P (P fixed , T varied)
  - Ex: Query P in database T
  - Three Algorithms:
    - Finite Automata
    - Knuth – Morris - Pratt
    - Boyer – Moore
- Preprocessing on T (T fixed , P varied)
  - Ex: Look for P in dictionary T
  - Algorithm: Suffix Tree

Rabin-Karp

- Although worst case behavior is O((n-m+1)m) the average case behavior of this algorithm is very good
- For illustrative purposes we will use radix 10 and decimal digits, but the arguments easily extend to other character set bases
- We associate a numeric value with every pattern

$$p = P[m] + 10\,(P[m-1] + 10(P[m-2] + \cdots + 10(P[2] + 10P[1])\cdots))$$

  - using Horner’s rule this is calculated in O(m) time
  - in a similar manner T[1..m] can be calculated in O(m)
  - if these values are equal then the strings match

♦ t_s =decimal value associated with T[s+1, …, s+m]

Calculating Remaining Values

- We need a quick way to calculate t_{s+1} from the value t_s without "starting from scratch"

$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1]$$

♦ For example, if m = 5, t_s is 31415, T[s+1] = 3, and T[s+ 5 + 1] = 2 then

$$\begin{aligned} t_{s+1} &= 10(31415 - 10000 \cdot 3) + 2 \\ &= 14152 \end{aligned}$$

– assuming 10^{m-1} is a stored constant, this calculation can be done in constant time

– the calculations of p, $t_0, t_1, t_2, \ldots, t_{n-m}$ together can be done in O(n+m) time

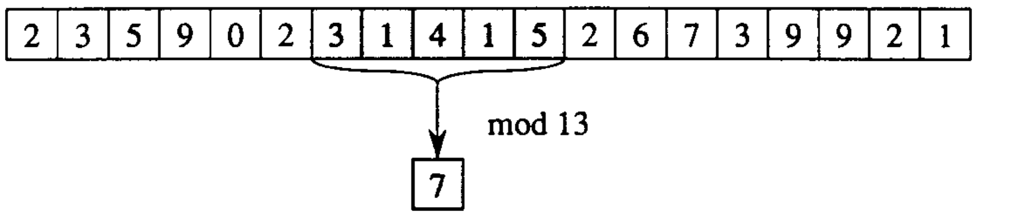
So What's the Problem

- Integer values may become too large
  - mod all calculations by a selected value, q
  - for a d-ary alphabet select q to be a large prime such that dq fits into one computer word

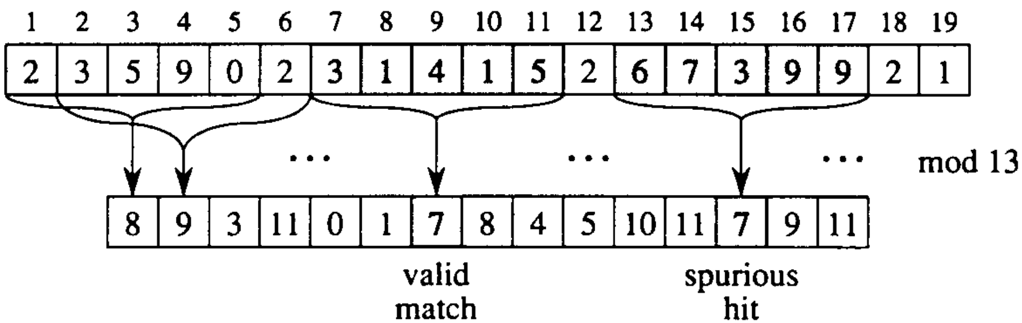
$$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q$$

- What if two values collide?
  - Similar to hash table functions, it is possible that two or more different strings produce the same "hit" value
  - any hit will have to be tested to verify that it is not spurious and that p[1..m] = T[s+1..s+m]

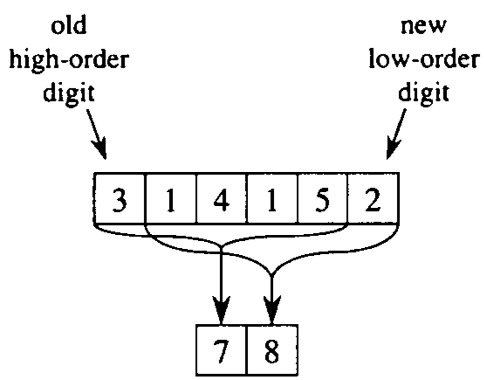
The Calculations



(a)



(b)



old high-order digit — shift — new low-order digit

$$
\begin{aligned}
14152 &\equiv (31415 - 3 \cdot 10000) \cdot 10 + 2 \pmod{13} \\
&\equiv (7 - 3 \cdot 3) \cdot 10 + 2 \pmod{13} \\
&\equiv 8 \pmod{13}
\end{aligned}
$$

(c)

The Algorithm

```
n ← length[T]
m ← length[P]
h ← d^(m−1) mod q
p ← 0
t_0 ← 0
for i ← 1 to m
     do p ← (dp + P[i]) mod q
        t_0 ← (dt_0 + T[i]) mod q
for s ← 0 to n − m
     do if p = t_s
           then if P[1 . . m] = T[s + 1 . . s + m]
                   then "Pattern occurs with shift" s
        if s < n − m
           then t_{s+1} ← (d(t_s − T[s + 1]h) + T[s + m + 1]) mod q
```

Complexity Analysis

- The worst case
  - every substring produces a hit
  - spurious checks are with the naïve algorithm, so the complexity is O((n-m+1)m)
- The average case
  - assume mappings from Σ^* to Z_q are random
  - we expect the number of spurious hits to be O(n/q)
  - the complexity is O(n) + O(m (v + n/q)) where v is the number of valid shifts
  - if q >= m then the running time is O(n+m)

Finite Automata

- A *finite automaton* M is a 5-tuple (Q, q_0, A, Σ, δ), where
  - Q is a finite set of *states*
  - $q_0 \varepsilon Q$ is the *start state*
  - $A \subseteq Q$ is a set of *accepting states*
  - Σ is a finite *input alphabet*
  - δ is the *transition function* that gives the next state for a given current state and input

How a Finite Automaton Works

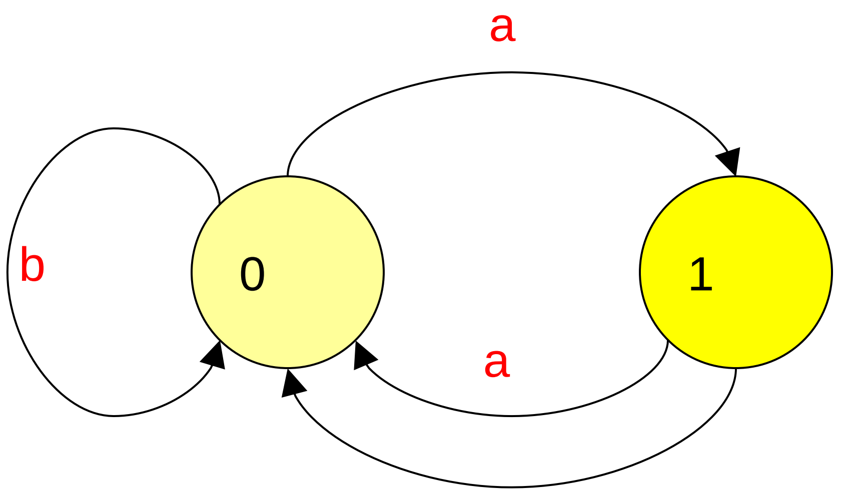
- The finite automaton M begins in state q_0
- Reads characters from Σ one at a time
- If M is in state q and reads input character a, M moves to state $\delta(q,a)$
- If its current state q is in A, M is said to have *accepted* the string read so far
- An input string that is not accepted is said to be *rejected*

Example

- $Q = \{0,1\}$, $q_0 = 0$, A={1}, $\Sigma = \{a, b\}$
- $\delta(q,a)$ shown in the transition table/diagram
- This accepts strings that end in an odd number of a's; e.g., abbaaa is accepted, aa is rejected

	input	
state	a	b
0	1	0
1	0	0

transition table



transition diagram

String-Matching Automata

- Given the pattern $P[1..m]$, build a finite automaton M
  - The state set is $Q=\{0, 1, 2, \ldots, m\}$
  - The start state is 0
  - The only accepting state is m

- Time to build M can be large if Σ is large

String-Matching Automata …contd

- Scan the text string $T[1..n]$ to find all occurrences of the pattern $P[1..m]$

- String matching is efficient: $\Theta(n)$
  - Each character is examined exactly once
  - Constant time for each character
- But …time to compute δ is $O(m|\Sigma|)$
  - δ Has $O(m|\Sigma|)$ entries

Algorithm

Input: Text string $T[1..n]$, δ and m

Result: All valid shifts displayed

FINITE-AUTOMATON-MATCHER (T, m, δ)

$n \leftarrow length[T]$

$q \leftarrow 0$

for $i \leftarrow 1$ **to** n

$q \leftarrow \delta(q, T[i])$

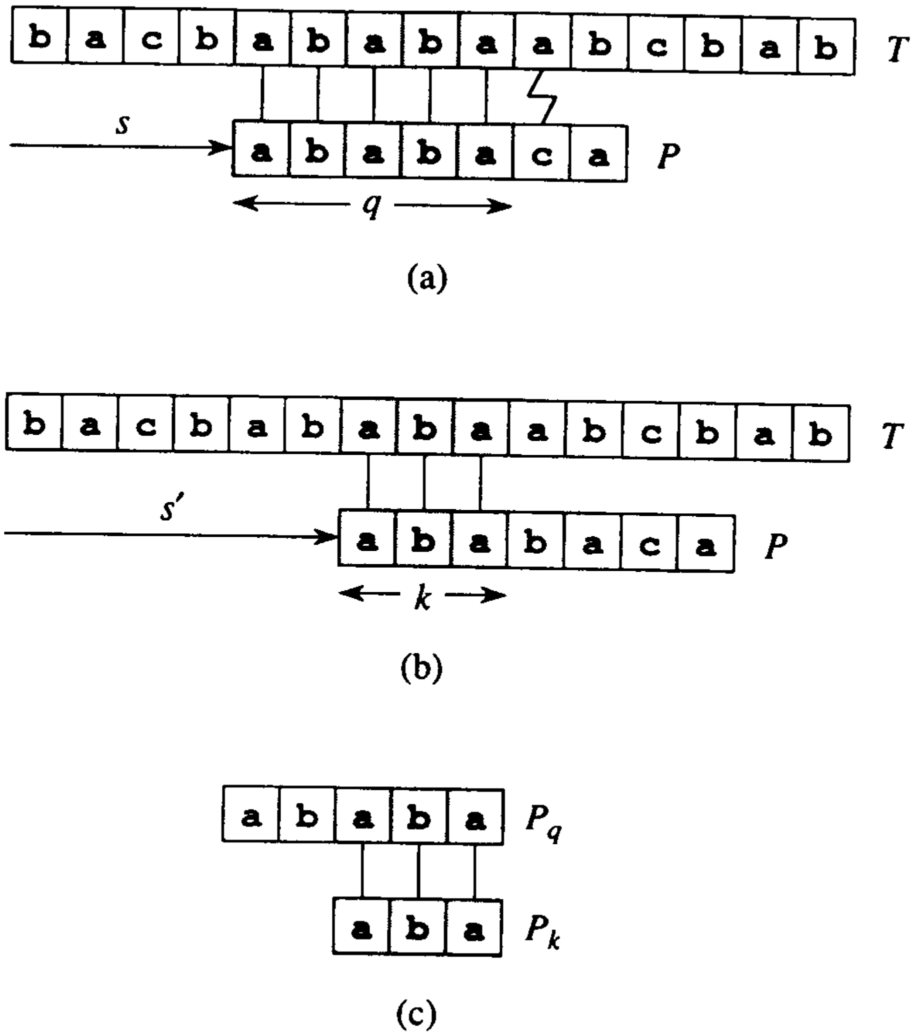
if $q = m$

print "pattern occurs with shift" $i\text{-}m$

Knuth-Morris-Pratt Algorithm

- The key observation
  - this approach is similar to the finite state automaton
  - when there is a mismatch after several characters match, then the pattern and search string contain the same values; therefore we can match the pattern against *itself* by precomputing a prefix function to find out how far we can shift ahead
  - this means we can dispense with computing the transition function δ altogether
- By using the prefix function the algorithm has running time of O(n + m)

Why Some Shifts are Invalid



- The first mismatch is at the 6th character
- consider the pattern already matched, it is clear a shift of 1 is not valid the beginning a in P would not match the b in the text
- the next valid shift is +2 because the aba in P matches the aba in the text
- the key insight is that we really only have to check the pattern matching against ITSELF

The prefix-function

- The question in terms of matching text

Given that pattern characters $P[1..q]$ match text characters $T[s+1..s+q]$, what is the least shift $s' > s$ such that

$$P[1..k] = T[s'+1..s'+k] , \tag{34.5}$$

where $s' + k = s + q$?

- The prefix-function in terms of the pattern

We formalize the precomputation required as follows. Given a pattern $P[1..m]$, the ***prefix function*** for the pattern P is the function $\pi : \{1, 2, \ldots, m\} \to \{0, 1, \ldots, m-1\}$ such that

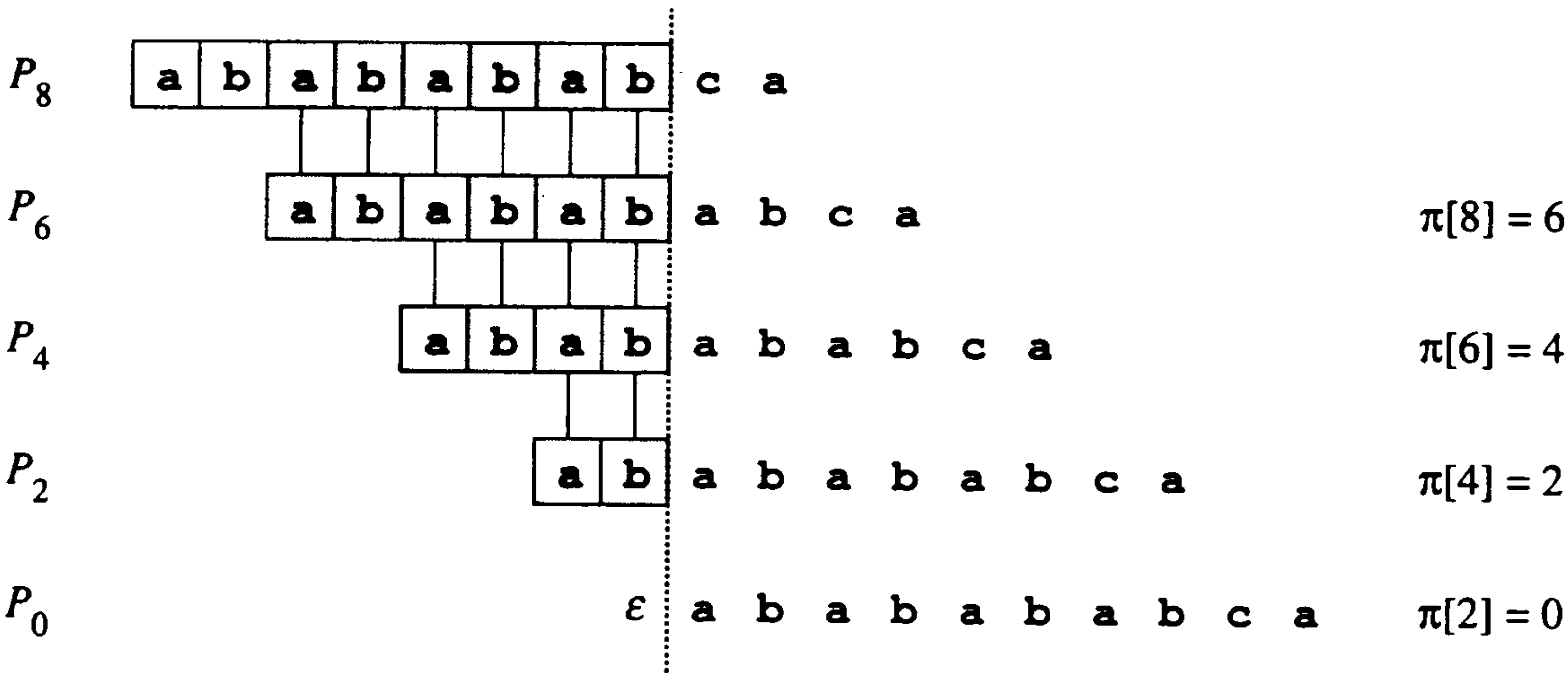
$$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\} .$$

- $\pi[q]$ is the length of the longest prefix of P that is a proper suffix of P_q

Another Example of the Prefix-function

i	1	2	3	4	5	6	7	8	9	10
$P[i]$	a	b	a	b	a	b	a	b	c	a
$\pi[i]$	0	0	1	2	3	4	5	6	0	1

(a)



(b)

Computing the Prefix-function

```
COMPUTE-PREFIX-FUNCTION(P)
m ← length[P]
π[1] ← 0
k ← 0
for q ← 2 to m
        do while k > 0 and P[k + 1] ≠ P[q]
                do k ← π[k]
           if P[k + 1] = P[q]
              then k ← k + 1
           π[q] ← k
return π
```

At the begining of each for iteration
k=π[q-1]

The Knuth_Morris_Pratt Algorithm

```
KMP-MATCHER(T, P)
n ← length[T]
m ← length[P]
π ← COMPUTE-PREFIX-FUNCTION(P)
q ← 0
for i ← 1 to n
        do while q > 0 and P[q + 1] ≠ T[i]
                do q ← π[q]
           if P[q + 1] = T[i]
             then q ← q + 1
           if q = m
             then print "Pattern occurs with shift" i − m
                  q ← π[q]
```

Runtime Analysis

- Calculations for the prefix function
  - we use an amortized analysis using the potential k
  - it is initially zero and always nonnegative, $\pi[k] >= 0$
  - the amortized cost of lines 5-9 is O(1)
  - since the outer loop is O(m) the worst case is O(m)
- Calculations for Knuth-Morris-Pratt
  - the call to compute prefix is O(m)
  - using q as the value of the potential function, we argue in the same manner as above to show the loop is O(n)
  - therefore the overall complexity is O(m + n)

Boyer-Moore Algorithm

- For longer patterns and large Σ it is the most efficient algorithm
- Some characteristics
  - it compares characters from right to left
  - it adds a “bad character” heuristic
  - it adds a “good suffix” heuristic
  - these two heuristics generate two different shift values; the larger of the two values is chosen
- In some cases Boyer-Moore can run in sub-linear time which means it may not be necessary to check all of the characters in the search text!

String
pattern matching - Boyer-Moore

Again, this algorithm uses fail-functions to shift the pattern efficiently. Boyer-Moore starts however at the end of the pattern, which can result in larger shifts.

Two heuristics are used:

1: if you encounter a mismatch at character *c* in Q, you can shift to the first occurrence of *c* in P from the right:

```
Q   a b c a b c d g a b c e a b c d a c e d

P   a b c e b c d

          a b c e b c d   (restart here)
```

String

pattern matching - Boyer-Moore

The first shift is precomputed and put into an alphabet-sized array fail1[] (complexity O(S), S=size of alphabet)

pattern: **a b c e b c d**

fail1:

a	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**...**
6	**2**	**1**	**0**	**3**	**7**	**7**	**7**	**7**	**7**	**...**

(shift = 6)

a b c a b c d g a b c e a b c d a c e d

a b c ~~e~~ b c d

a b c e b c d

String
pattern matching - Boyer-Moore

2: if the examined suffix of P occurs also as substring in P, a shift can be performed to the first occurrence of this substring:

c e a b c d a b c f g a b c e a b

d a b c ~~a~~ a b c

we can shift to: **d a b c a a b c** (restart here)

pattern: **d a b c a a b c**
fail2 **7**

String

pattern matching - Boyer-Moore

The second shift is also precomputed and put into a length(P)-sized array fail2[] (complexity O(length(P))

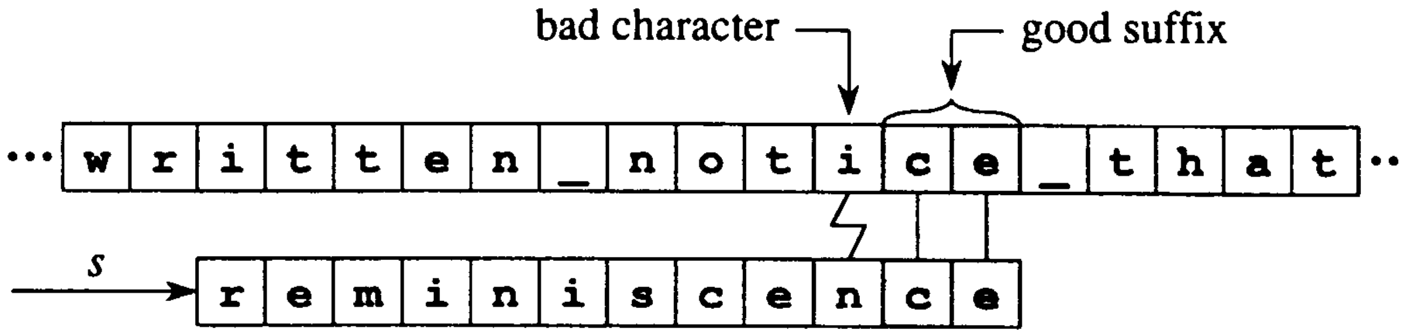
pattern:	**d**	**a**	**b**	**c**	**a**	**a**	**b**	**c**
fail2	**9**	**9**	**9**	**9**	**7**	**6**	**5**	**1**

(shift = 7)

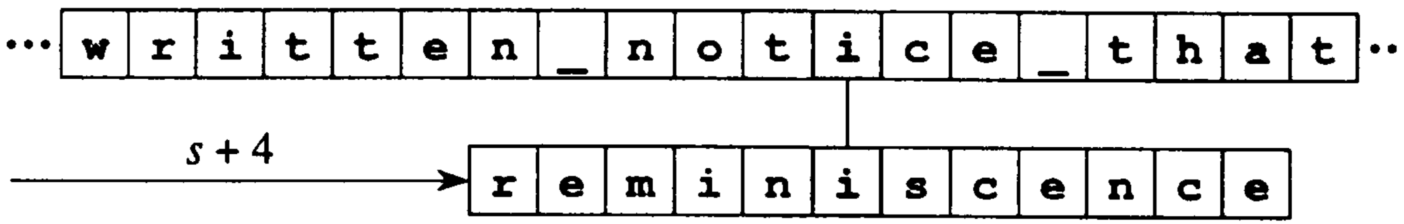
c e a b c d a b c f g a b c e a b

d a b c ~~a~~ a b c

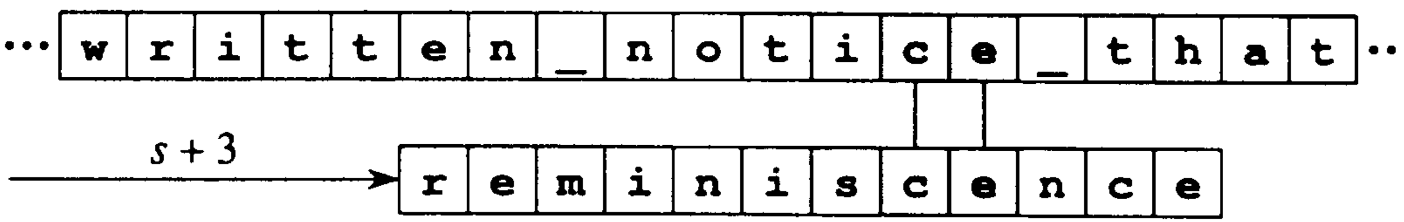
d a b c a a b c


bad character
good suffix
w r i t t e n _ n o t i c e _ t h a t
s
r e m i n i s c e n c e

(a)


w r i t t e n _ n o t i c e _ t h a t
s + 4
r e m i n i s c e n c e

(b)


w r i t t e n _ n o t i c e _ t h a t
s + 3
r e m i n i s c e n c e

(c)

```
BOYER-MOORE-MATCHER(T, P, Σ)
 1  n ← length[T]
 2  m ← length[P]
 3  λ ← COMPUTE-LAST-OCCURRENCE-FUNCTION(P, m, Σ)
 4  γ ← COMPUTE-GOOD-SUFFIX-FUNCTION(P, m)
 5  s ← 0
 6  while s ≤ n − m
 7       do j ← m
 8          while j > 0 and P[j] = T[s + j]
 9              do j ← j − 1
10          if j = 0
11            then print "Pattern occurs at shift" s
12                 s ← s + γ[0]
13            else s ← s + max(γ[j], j − λ[T[s + j]])
```

- If lines 12 and 13 were changed to

```
12                 s ← s + 1
13            else s ← s + 1
```

then we would have a naïve-string matcher

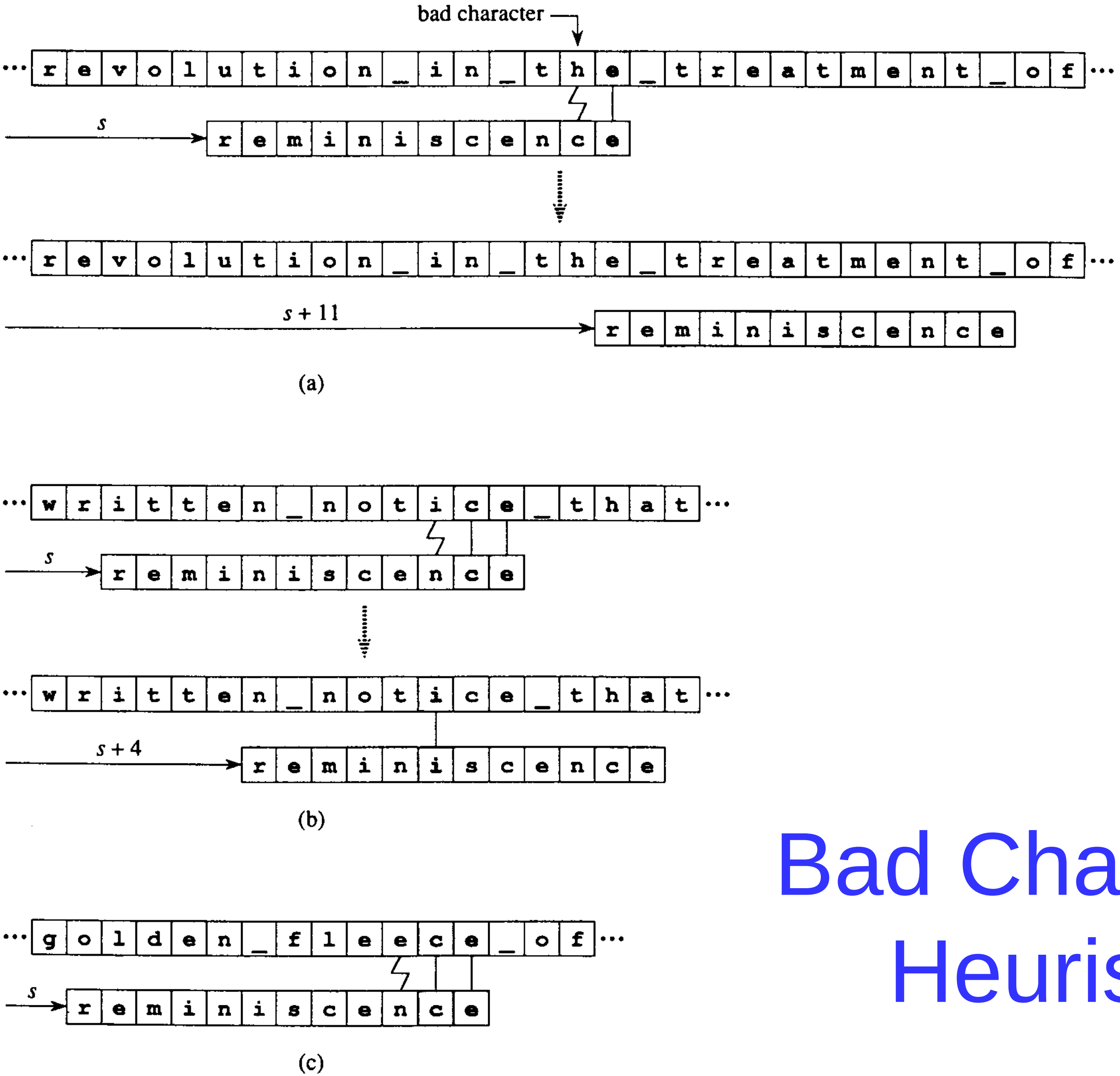
Bad Character Heuristic

- Best case behavior
  - the rightmost character causes a mismatch
  - the character in the text does not occur anywhere in the pattern
  - therefore the entire pattern may be shifted past the bad character and many characters in the text are not examined at all
- This illustrates the benefit of searching from right to left as opposed to left to right
- This is the first of three cases we have to consider in generating the bad character heuristic

Bad Character Heuristic

- Assume $P[j] \neq T[s+j]$ and k is the largest index such that $P[k] = T[s+j]$
- Case 2 - the first occurrence is to the left of the mismatch
  - $k < j$ so $j-k > 0$
  - it is safe to increase s by j-k without missing any valid shifts
- Case 3 - the first occurrence is to the right of the mismatch
  - $k > j$ so $j-k < 0$
  - this proposes a negative shift, but the Boyer-Moore algorithm ignores this “advice” since the good suffix heuristic always proposes a shift of 1 or more

Bad Character Heuristic



The Last Occurrence Function

- The function λ(ch) finds the rightmost occurrence of the character ch inside P

COMPUTE-LAST-OCCURRENCE-FUNCTION(P, m, Σ)

for each character $a \in \Sigma$
    do $\lambda[a] = 0$
for $j \leftarrow 1$ **to** m
    do $\lambda[P[j]] \leftarrow j$
return λ

♦ The complexity of this function is $O(|\Sigma| + m)$

The Good-suffix Heuristic

- It is shown that the prefix function can be used to simplify γ[j]

$$
\begin{aligned}
\gamma[j] &= m - \max(\{\pi[m]\} \\
&\qquad\qquad \cup \{m - l + \pi'[l] : 1 \le l \le m \text{ and } j = m - \pi'[l]\}) \\
&= \min(\{m - \pi[m]\} \\
&\qquad\qquad \cup \{l - \pi'[l] : 1 \le l \le m \text{ and } j = m - \pi'[l]\}) \ . \qquad (34.9)
\end{aligned}
$$

- π’ is derived from the reversed pattern P’

The Good-suffix Heuristic

- It is shown that the prefix function can be used to simplify γ[j]

$$
\begin{aligned}
\gamma[j] &= m - \max(\{\pi[m]\} \\
&\qquad \cup \{m - l + \pi'[l] : 1 \le l \le m \text{ and } j = m - \pi'[l]\}) \\
&= \min(\{m - \pi[m]\} \\
&\qquad \cup \{l - \pi'[l] : 1 \le l \le m \text{ and } j = m - \pi'[l]\}) . \qquad (34.9)
\end{aligned}
$$

COMPUTE-GOOD-SUFFIX-FUNCTION(P, m)

$\pi \leftarrow$ COMPUTE-PREFIX-FUNCTION(P)
$P' \leftarrow$ reverse(P)
$\pi' \leftarrow$ COMPUTE-PREFIX-FUNCTION(P')
for $j \leftarrow 0$ **to** m
    do $\gamma[j] \leftarrow m - \pi[m]$
for $l \leftarrow 1$ **to** m
    do $j \leftarrow m - \pi'[l]$
        if $\gamma[j] > l - \pi'[l]$
            then $\gamma[j] \leftarrow l - \pi'[l]$
return γ

Runtime Complexity

- The complexity of the heuristic functions are $O(m+|\Sigma|)$
- The worst case behavior of Boyer-Moore is $O((n-m+1)m + |\Sigma|)$, similar to naïve algorithm
- But, the actual behavior in practice is much better

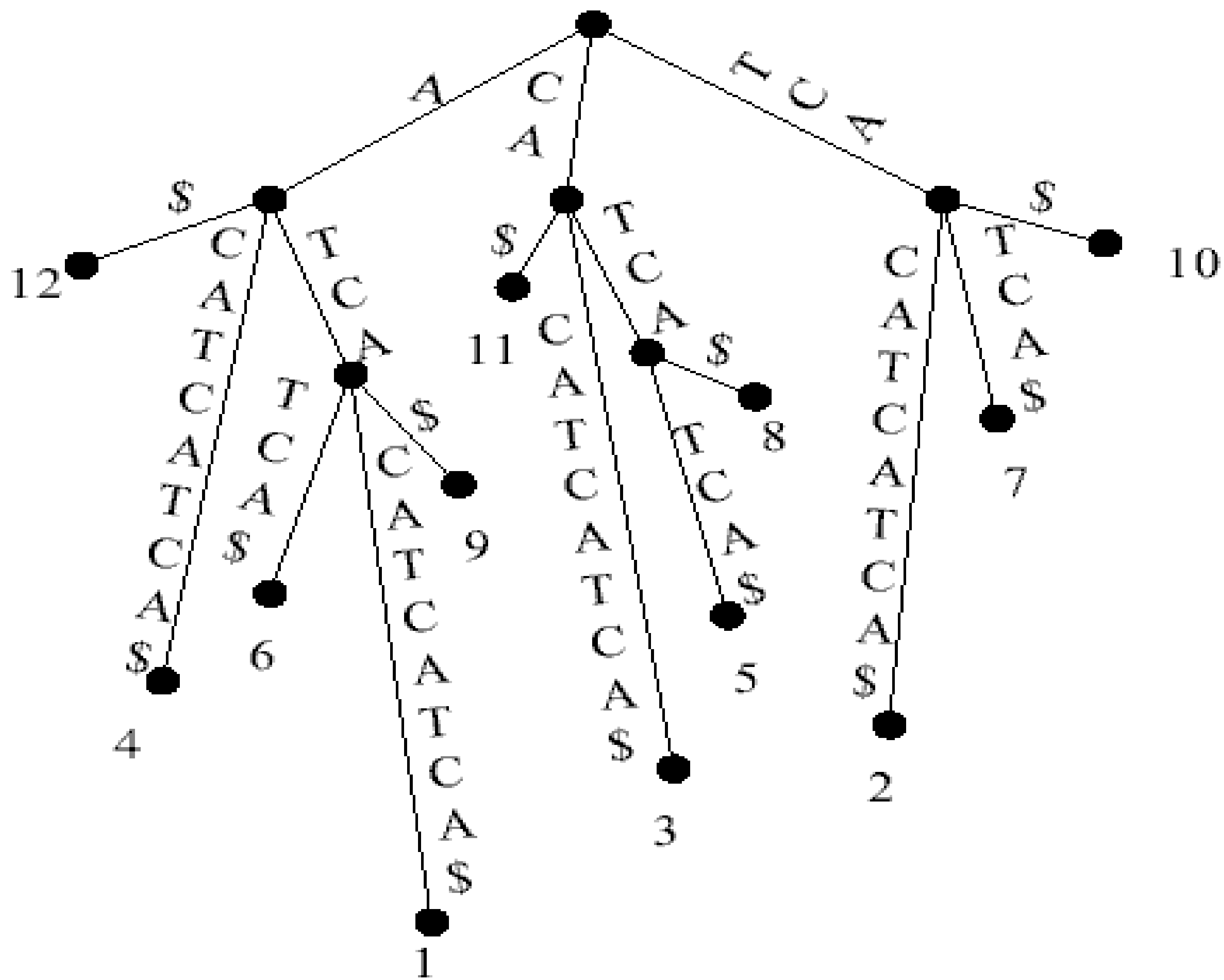
Suffixes

- Suffixes for S="ATCACATCATCA"

ATCACATCATCA	$S_{(1)}$
TCACATCATCA	$S_{(2)}$
CACATCATCA	$S_{(3)}$
ACATCATCA	$S_{(4)}$
CATCATCA	$S_{(5)}$
ATCATCA	$S_{(6)}$
TCATCA	$S_{(7)}$
CATCA	$S_{(8)}$
ATCA	$S_{(9)}$
TCA	$S_{(10)}$
CA	$S_{(11)}$
A	$S_{(12)}$

Suffix Trees

- A suffix Tree for S="ATCACATCATCA"



Properties of a Suffix Tree

- Each tree edge is labeled by a substring of S.
- Each internal node has at least 2 children.
- Each $S_{(i)}$ has its corresponding labeled path from root to a leaf, for $1 \leq i \leq n$.
- There are n leaves.
- No edges branching out from the same internal node can start with the same character.

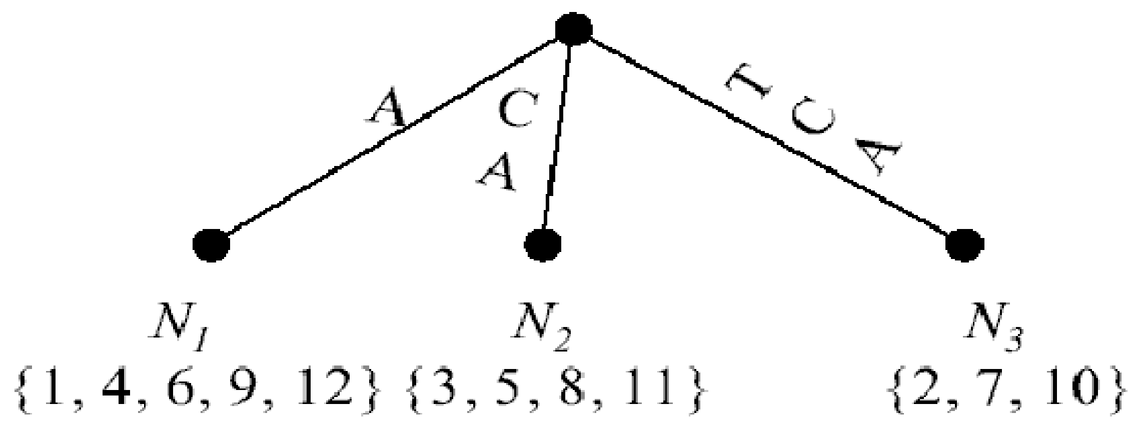
Algorithm for Creating a Suffix Tree

Step 1: Divide all suffixes into distinct groups according to their starting characters and create a node.

Step 2: For each group, if it contains only one suffix, create a leaf node and a branch with this suffix as its label; otherwise, find the longest common prefix among all suffixes of this group and create a branch out of the node with this longest common prefix as its label. Delete this prefix from all suffixes of the group.

Step 3: Repeat the above procedure for each node which is not terminated.

Example for Creating a Suffix Tree



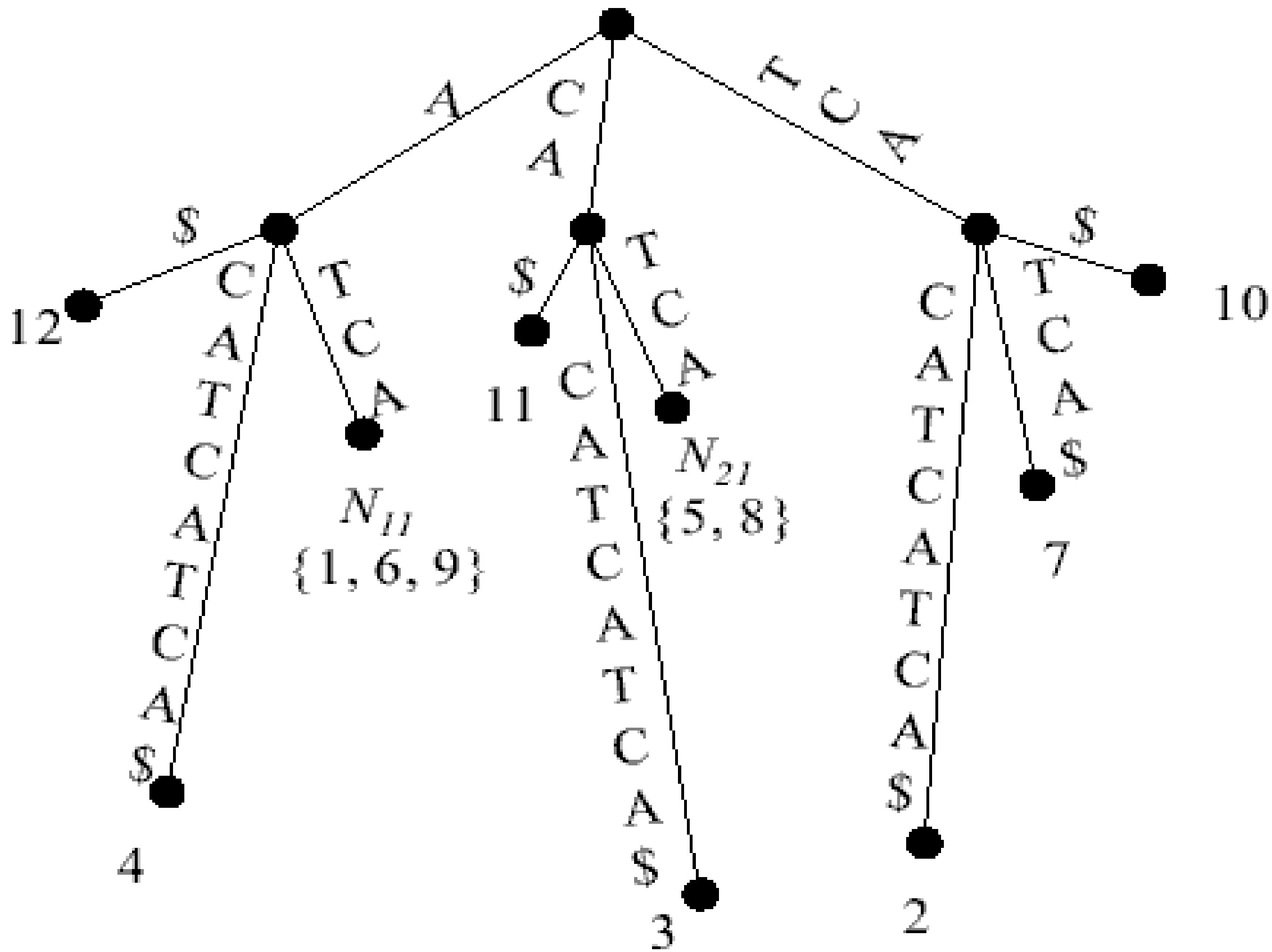
- S="ATCACATCATCA".
- Starting characters: "A", "C", "T"
- In $\boldsymbol{N_3}$,

  $S(2)$ ="TCACATCATCA"

  $S(7)$ ="TCATCA"

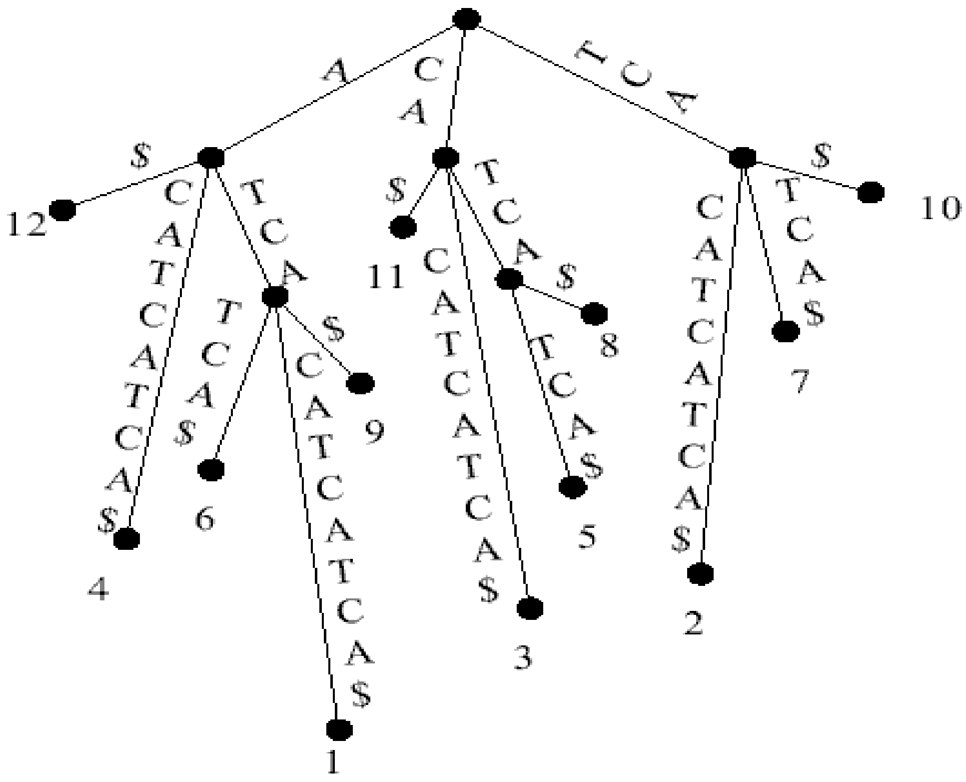
  $S(10)$ ="TCA"
- Longest common prefix of $\boldsymbol{N_3}$ is "TCA"

- S="ATCACATCATCA".
- Second recursion:



Finding a Substring with the Suffix Tree

- S = “ATCACATCATCA”
- P=“TCAT”
  - P is at position 7 in S.
- P=“TCA”
  - P is at position 2, 7 and 10 in S.
- P=“TCATT”
  - P is not in S.



Time Complexity

- A suffix tree for a text string T of length n can be constructed in O(n) time (with a complicated algorithm).
- To search a pattern P of length m on a suffix tree needs O(m) comparisons.
- Exact string matching: O(n+m) time

The Suffix Array

- In a suffix array, all suffixes of S are in the non-decreasing lexical order.
- For example, S="`ATCACATCATCA`"

i	1	2	3	4	5	6	7	8	9	10	11	12
A	12	4	9	1	6	11	3	8	5	10	2	7

4	`ATCACATCATCA`	$S_{(1)}$
11	`TCACATCATCA`	$S_{(2)}$
7	`CACATCATCA`	$S_{(3)}$
2	`ACATCATCA`	$S_{(4)}$
9	`CATCATCA`	$S_{(5)}$
5	`ATCATCA`	$S_{(6)}$
12	`TCATCA`	$S_{(7)}$
8	`CATCA`	$S_{(8)}$
3	`ATCA`	$S_{(9)}$
10	`TCA`	$S_{(10)}$
6	`CA`	$S_{(11)}$
1	`A`	$S_{(12)}$

1	`A`	$S_{(12)}$
2	`ACATCATCA`	$S_{(4)}$
3	`ATCA`	$S_{(9)}$
4	`ATCACATCATCA`	$S_{(1)}$
5	`ATCATCA`	$S_{(6)}$
6	`CA`	$S_{(11)}$
7	`CACATCATCA`	$S_{(3)}$
8	`CATCA`	$S_{(8)}$
9	`CATCATCA`	$S_{(5)}$
10	`TCA`	$S_{(10)}$
11	`TCACATCATCA`	$S_{(2)}$
12	`TCATCA`	$S_{(7)}$

Searching in a Suffix Array

- If T is represented by a suffix array, we can find P in T in $O(m\log n)$ time with a binary search.
- A suffix array can be determined in $O(n)$ time by lexical depth first searching in a suffix tree.
- Total time: $O(n+m\log n)$