Lect08. String Matching

String Matching Problem

- Given a text string *T* of length *n* and a pattern string *P* of length *m*, the exact string matching problem is to find all occurrences of *P* in *T*.
- Example: T="AGCTTGA" P="GCT"
- Applications:
  - Searching keywords in a file
  - Searching engines (like Google and Openfind)
  - Database searching (GenBank)
- More string matching algorithms (with source codes):

  http://www-igm.univ-mlv.fr/~lecroq/string/

String Matching

- 입력
  - A[1…n]: 텍스트 문자열
  - P[1…m]: 패턴 문자열
  - $m << n$
- 수행 작업
  - 텍스트 문자열 A[1…n]이 패턴 문자열 P[1…m]을 포함하는지 알아본다

Terminologies

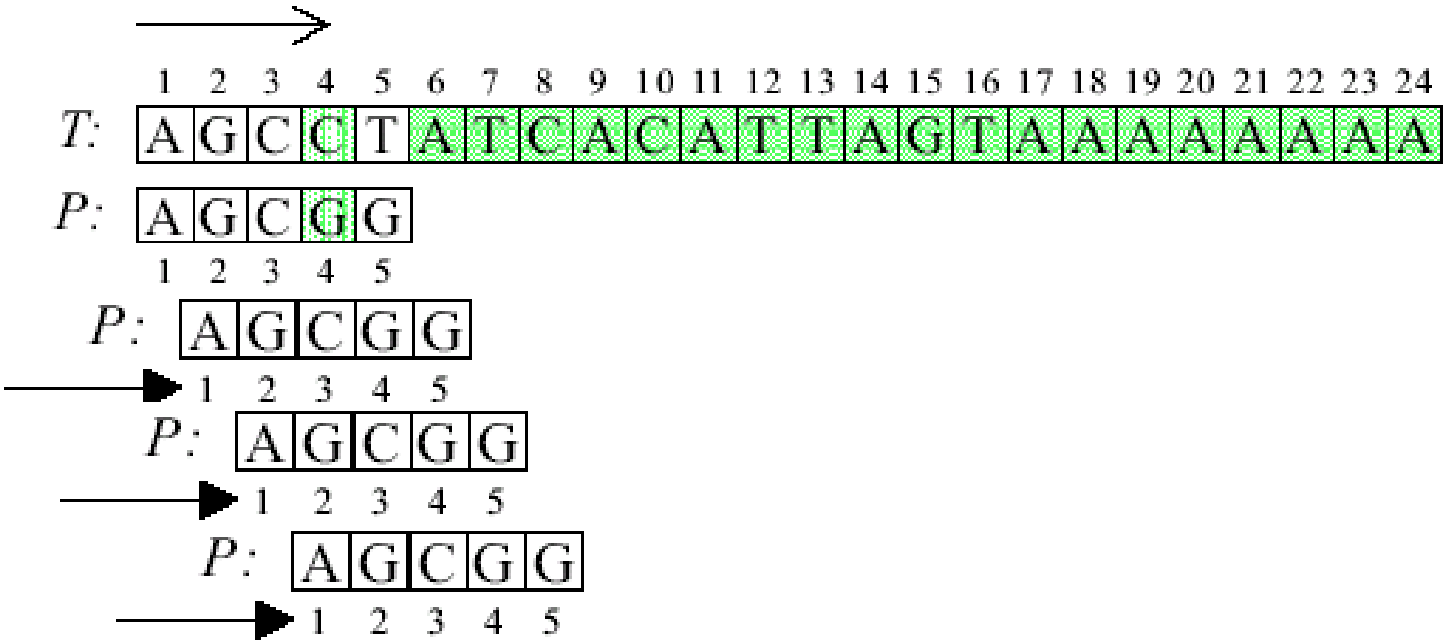
- S="AGCTTGA"
- |S|=7, length of S
- Substring: $S_{i,j} = S_i S_{i+1} \dots S_j$
  - Example: $S_{2,4}$="GCT"
- Subsequence of S: deleting zero or more characters from S
  - "ACT" and "GCTT" are subsquences.
- Prefix of S: $S_{1,k}$
  - "AGCT" is a prefix of S.
- Suffix of S: Sh, |S|
  - "CTTGA" is a suffix of S.

A Brute-Force Algorithm

```
naiveMatching(A[ ], P[ ])
{
 ▷ n: 배열 A[ ]의 길이, m: 배열 P[ ]의 길이
 for i ← 1 to n-m+1{
                    if (P[1…m] = A[i…i+m-1]) then
                      A[i] 자리에서 매칭이 발견되었음을 알린다;
 }
}
```

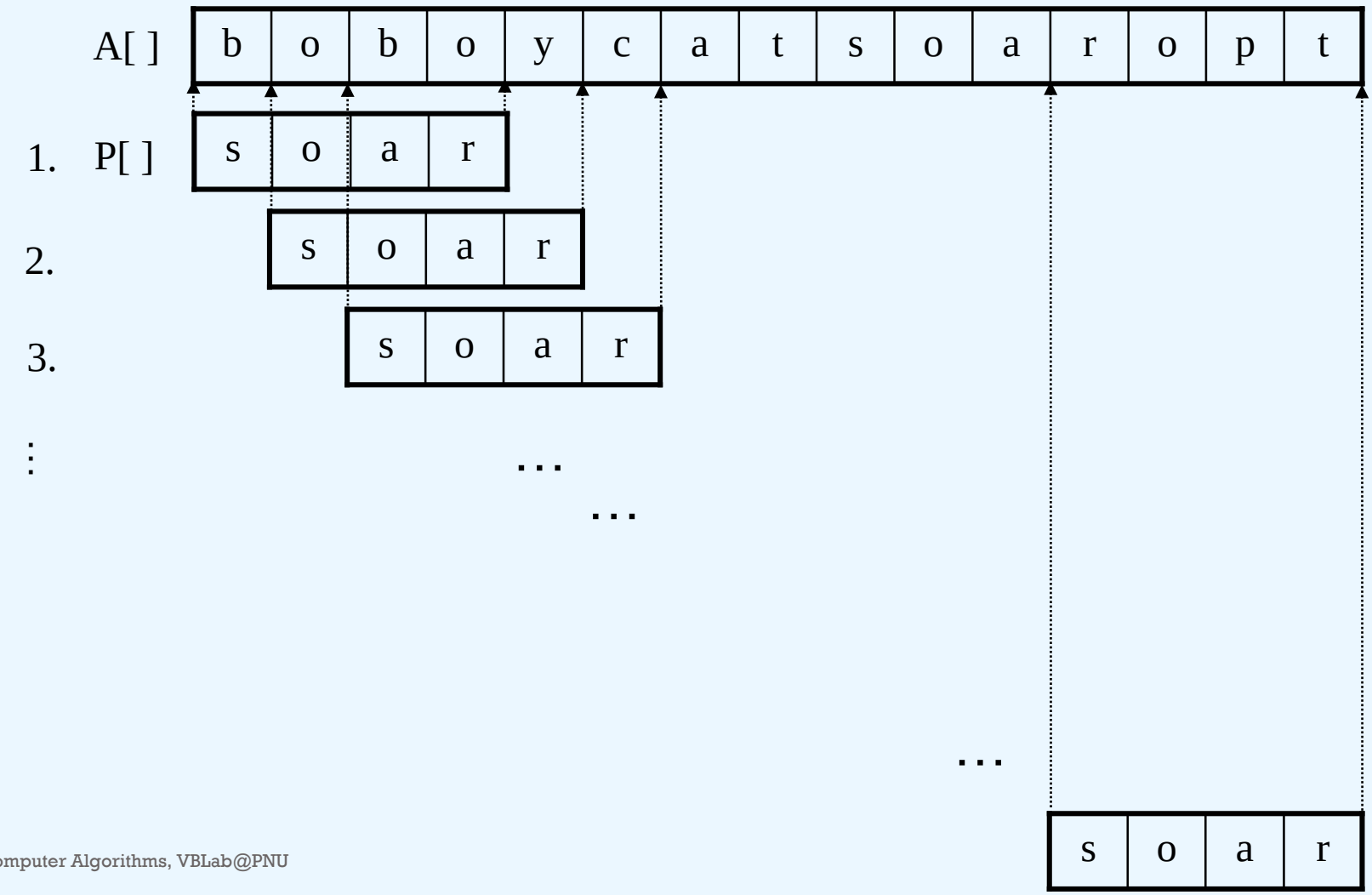
✓ 수행시간: $O(mn)$

A Brute-Force Algorithm

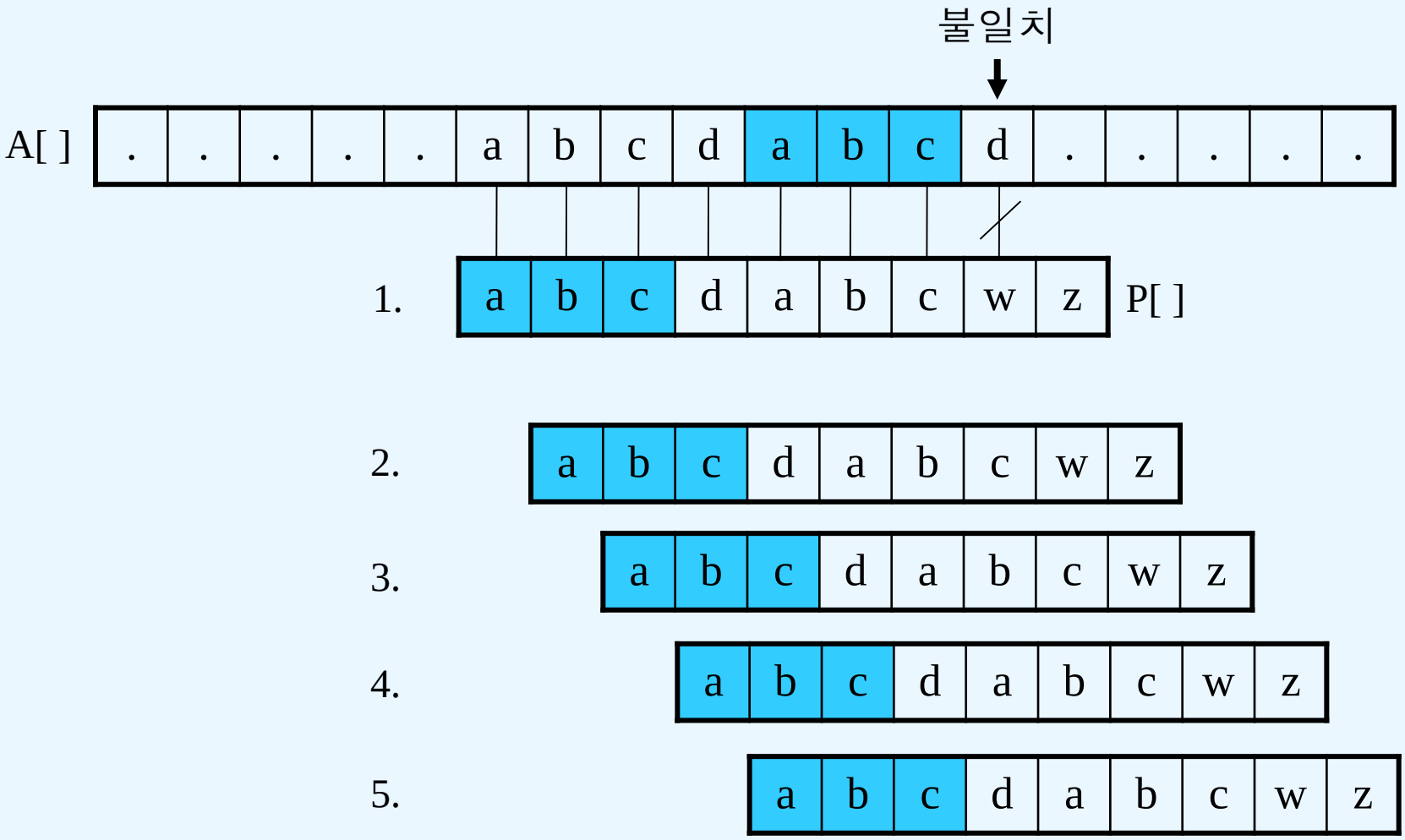


Time: O(mn) where m=|P| and n=|T|.

원시적인 매칭의 작동원리



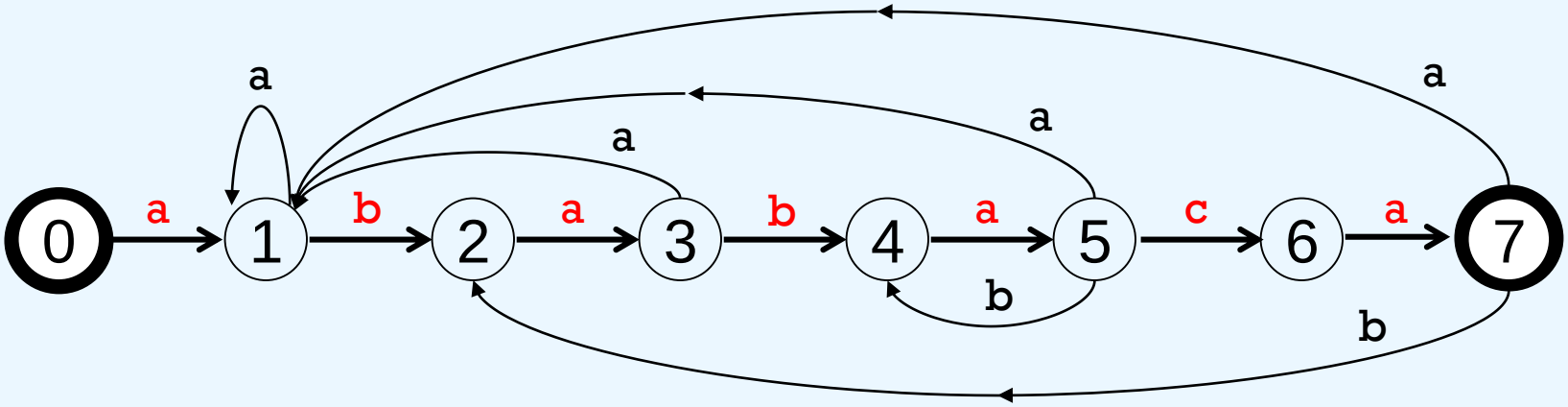
원시적인 매칭이 비효율적인 예



오토마타를 이용한 매칭

- 오토마타
  - 문제 해결 절차를 상태state의 전이로 나타낸 것
  - 구성 요소: (Q, q0, A, ∑, δ)
    - Q : 상태 집합
    - q0 : 시작 상태
    - A : 목표 상태들의 집합
    - ∑ : 입력 알파벳
    - δ : 상태 전이 함수
- 매칭이 진행된 상태들간의 관계를 오토마타로 표현한다

ababaca를 체크하는 오토마타

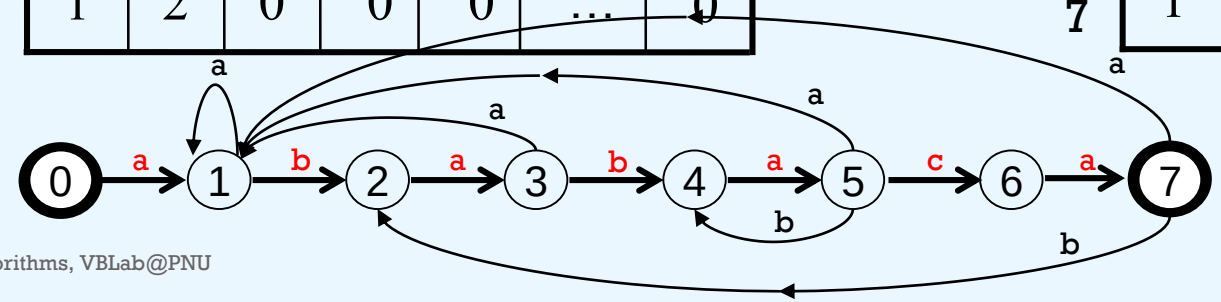


S: dvganbbactababaababacabababacaagbk…

오토마타의 S/W 구현

상태 \ 입력문자	a	b	c	d	e	…	z
0	1	0	0	0	0	…	0
1	1	2	0	0	0	…	0
2	3	0	0	0	0	…	0
3	1	4	0	0	0	…	0
4	5	0	0	0	0	…	0
5	1	4	6	0	0	…	0
6	7	0	0	0	0	…	0
7	1	2	0	0	0	…	0

상태 \ 입력문자	a	b	c	기타
0	1	0	0	0
1	1	2	0	0
2	3	0	0	0
3	1	4	0	0
4	5	0	0	0
5	1	4	6	0
6	7	0	0	0
7	1	2	0	0



오토마타를 이용해 매칭을 체크하는 알고리즘

```
FA-Matcher (A, δ, f)
▷ f: 목표 상태
{
    ▷ n: 배열 A[ ]의 길이
    q ← 0;
    for i ← 1 to n {
        q ← δ(q, A[i]);
        if (q = f) then A[i-m+1]에서 매칭이 발생했음을 알린다;
    }
}
```

구성 요소: (Q, q0, A, Σ, δ)
Q : 상태 집합
q0 : 시작 상태
A : 목표 상태들의 집합
Σ : 입력 알파벳
δ : 상태 전이 함수

✓ 총 수행시간: $\Theta(n + |\sum|m)$

Rabin-Karp Algorithm

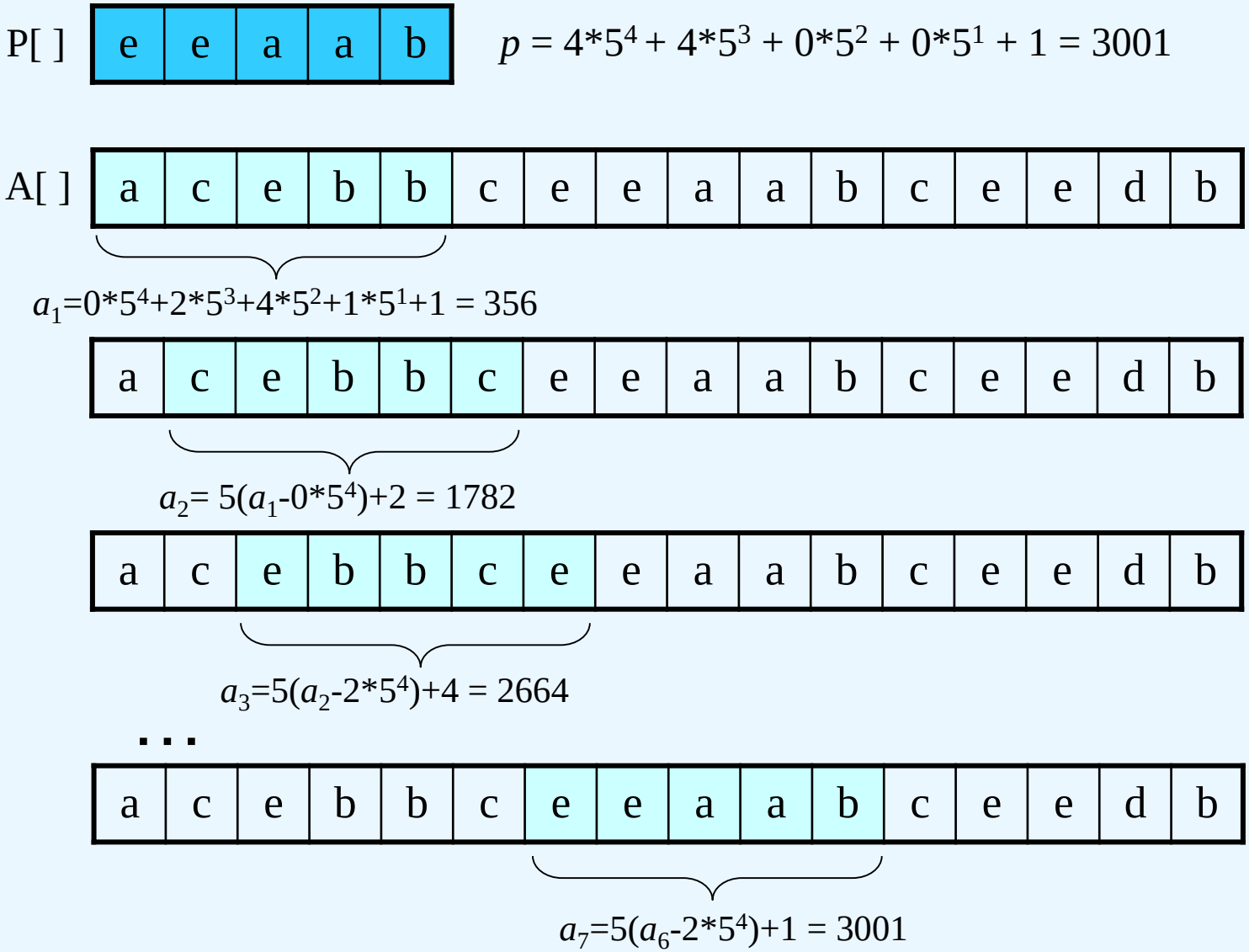
Rabin-Karp Algorithm

- 문자열 패턴을 수치로 바꾸어(즉 해시값) 문자열의 비교를 수치 비교로 대신한다
- 수치화
  - 가능한 문자 집합 $\sum$의 크기에 따라 진수가 결정된다
  - 예: $\sum$ = {a, b, c, d, e}
    - $|\sum| = 5$
    - a, b, c, d, e를 각각 0, 1, 2, 3, 4에 대응시킨다
    - 문자열 “cad”를 수치화하면 $2*5^2 + 0*5^1 + 3*5^0 = 28$

수치화 작업의 부담

- A[i…i+m-1]에 대응되는 수치의 계산
  - $a_i = \mathrm{A}[i+m-1] + d(\mathrm{A}[i+m-2] + d(\mathrm{A}[i+m-3] + d(\ldots + d(\mathrm{A}[i]))\ldots)$
  - $\Theta(m)$의 시간이 든다
  - 그러므로 A[1…n] 전체에 대한 비교는 $\Theta(mn)$이 소요된다
  - 원시적인 매칭에 비해 나은 게 없다
- 다행히,
  m의 크기에 상관없이 아래와 같이 계산할 수 있다
  - $a_i = d(a_{i-1} - d^{m-1}\mathrm{A}[i-1]) + \mathrm{A}[i+m-1]$
  - d^{m-1}은 반복 사용되므로 미리 한번만 계산해 두면 된다
  - 곱셈 2회, 덧셈 2회로 충분

수치화를 이용한 매칭의 예



. . .

수치화를 이용해 매칭을 체크하는 알고리즘

basicRabinKarp(A, P, d, q)
{
▷ n : 배열 A[ ]의 길이, m : 배열 P[ ]의 길이
$p \leftarrow 0$; $a_1 \leftarrow 0$;
for $i \leftarrow 1$ **to** m { ▷ a_1 계산
$p \leftarrow dp + \mathrm{P}[i]$;
$a_1 \leftarrow da_1 + \mathrm{A}[i]$;
}
for $i \leftarrow 1$ **to** n-m+1{
if ($i \neq 1$) **then** $a_i \leftarrow d(a_{i-1} - d^{m-1}\mathrm{A}[i\text{-}1]) + \mathrm{A}[i+m\text{-}1]$;
if ($p = a_i$) **then** A[i] 자리에서 매칭이 되었음을 알린다;
}
}

✓ 총 수행시간: $\Theta(n)$

앞의 알고리즘의 문제점

- 문자 집합 Σ와 m의 크기에 따라 a_i가 매우 커질 수 있다
  - 심하면 컴퓨터 레지스터의 용량 초과
  - 오버플로우 발생
- 해결책
  - 나머지 연산modulo을 사용하여 a_i의 크기를 제한한다
  - $a_i = d(a_{i-1} - d^{m-1}\mathrm{A}[i\text{-}1]) + \mathrm{A}[i+m\text{-}1]$ 대신
    $b_i = (d(b_{i-1} - (d^{m-1} \bmod q)\,\mathrm{A}[i\text{-}1]) + \mathrm{A}[i+m\text{-}1]) \bmod q$ 사용
  - q를 충분히 큰 소수로 잡되, dq가 레지스터에 수용될 수 있도록 잡는다

나머지 연산을 이용한 매칭의 예

P[ ] | e | e | a | a | b |

$p = (4*5^4 + 4*5^3 + 0*5^2 + 0*5^1 + 1) \bmod 113 = 63$

A[ ] | a | c | e | b | b | c | e | e | a | a | b | c | e | e | d | b |

$a_1 = (0*5^4+2*5^3+4*5^2+1*5^1+1) \bmod 113 = 17$

| a | c | e | b | b | c | e | e | a | a | b | c | e | e | d | b |

$a_2 = (5(a_1 - 0*(5^4 \bmod 113))+2) \bmod 113 = 87$

| a | c | e | b | b | c | e | e | a | a | b | c | e | e | d | b |

$a_3 = (5(a_2 - 2*(5^4 \bmod 113))+4) \bmod 113 = 65$

...

| a | c | e | b | b | c | e | e | a | a | b | c | e | e | d | b |

...

$a_7 = (5(a_6 - 2*(5^4 \bmod 113))+1) \bmod 113 = 63$

Rabin-Karp Algorithm

```
RabinKarp(A, P, d, q)
{
    ▷ n : 배열 A[ ]의 길이, m : 배열 P[ ]의 길이
    p ← 0; b1 ← 0;
    for i ← 1 to m {                                    ▷ b1 계산
        p ← (dp + P[i]) mod q;
        b1 ← (db1 + A[i]) mod q;
    }
    h ← d^(m-1) mod q;
    for i ← 1 to n-m+1 {
        if (i ≠ 1) then bi ← (d(bi-1 – hA[i-1]) + A[i+m-1]) mod q;
        if (p = bi) then
            if (P[1…m] = A[i…i+m-1]) then
                A[i] 자리에서 매칭이 되었음을 알린다;
    }
}
```

✓ 평균 수행시간: $\Theta(n)$

Analysis of Rabin-Karp Algorithm

- 패턴의 해시값과 텍스트의 해시 값이 일치하는 경우 다시 일일이 문자 비교를 해야 하기 때문에 최악의 실행시간은 **O(mn)**

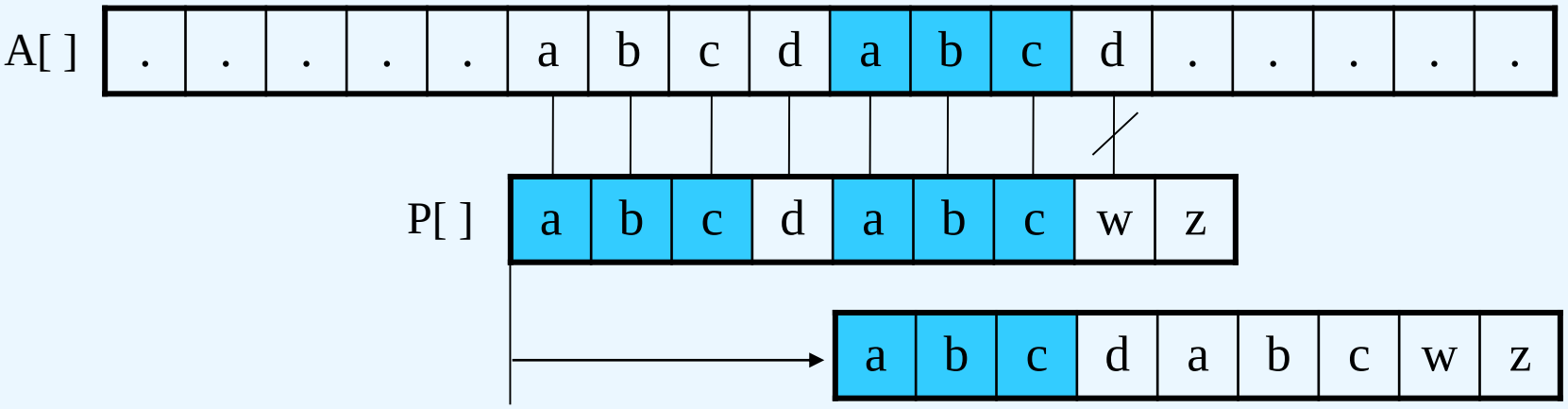
  $P=a^m, T=a^n$

- 패턴의 해시값과 텍스트의 해시값이 모두 불일치하는 최선의 경우 **O(m+n)**시간
- 실제로 대부분의 경우 해시값이 불일치하는 경우가 많을 것이며 따라서 **O(m+n)**시간에 실행될 가능성이 높음

KMP Algorithm

KMP Knuth-Morris-Pratt Algorithm

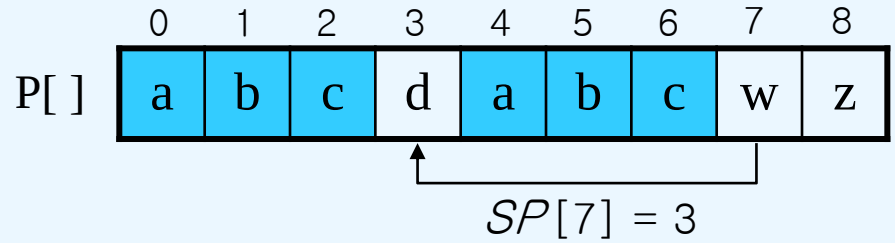
- Knuth, Morris, Pratt 세 사람이 고안
- 패턴을 전처리(preprocessing)하여 접미부와 일치하는 최대 접두부(SP)를 구하고 이 정보를 이용하여 매칭하는 방법
- 오토마타를 이용한 매칭과 동기가 유사
- 공통점
  - 매칭에 실패했을 때 돌아갈 상태를 준비해둔다
  - 오토마타를 이용한 매칭보다 준비 작업이 단순하다
- O(n+ m)시간 알고리즘



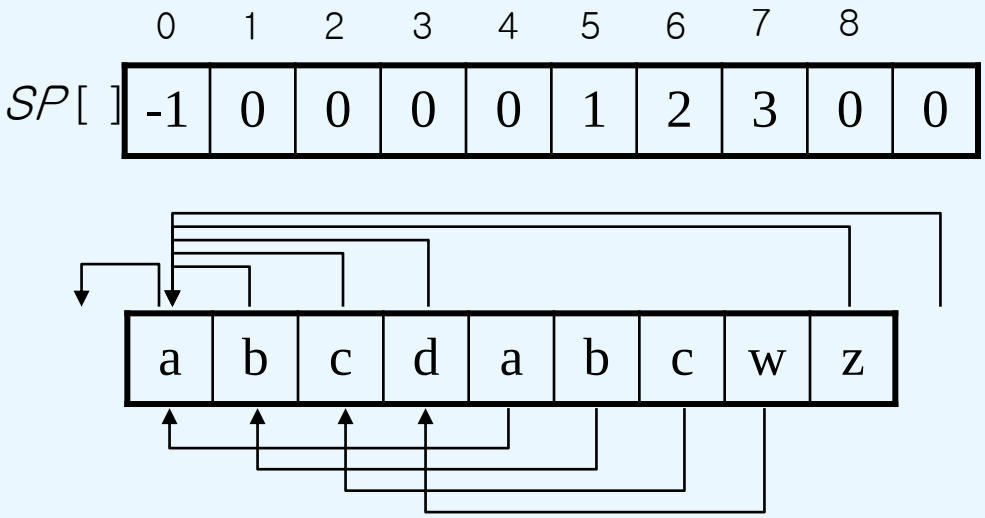
```
T: a b a b a b a c a b a b a b c a c a a b b c
P: a b a b a b c a
T: a b a b a b a c a b a b a b c a c a a b b c
P:   a b a b a b c a
T: a b a b a b a c a b a b a b c a c a a b b c
P:     a b a b a b c a
T: a b a b a b a c a b a b a b c a c a a b b c
P:       a b a b a b c a
T: a b a b a b a c a b a b a b c a c a a b b c
P:         a b a b a b c a
T: a b a b a b a c a b a b a b c a c a a b b c
P:           a b a b a b c a
```

$SP_k : P_k \supset P_i$ 인 최대 P_k, 이때 P_k는 $P[0 \ldots k]$. 최대접두부

매칭이 실패했을 때 돌아갈 곳 준비 작업



텍스트에서 abcdabc까지는 매치되고, w에서 실패한 상황
패턴의 맨앞의 abc와 실패 직전의 abc는 동일함을 이용할 수 있다
실패한 텍스트 문자와 P[3]를 비교한다

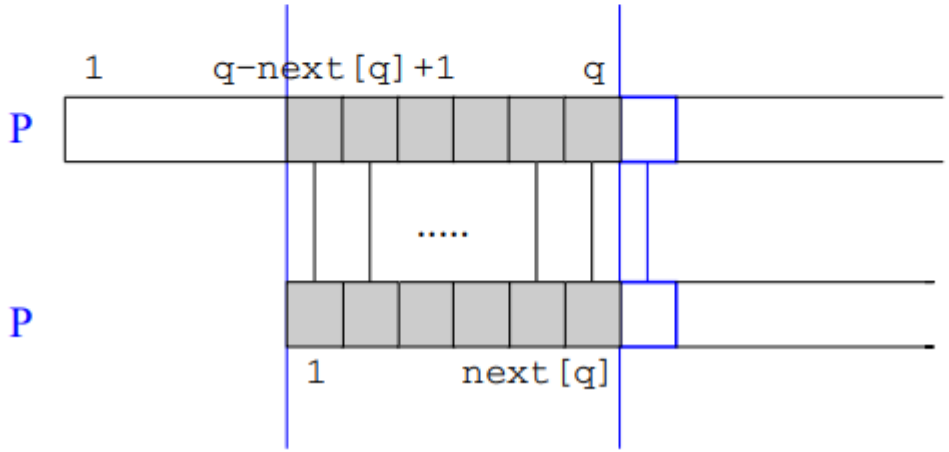


패턴의 각 위치에 대해
매칭에 실패했을 때
돌아갈 곳을 준비해 둔다

How to compute next[]/SP[] ?

Given `next[1], next[2], ..., next[q]`, how can we compute `next[q+1]`?

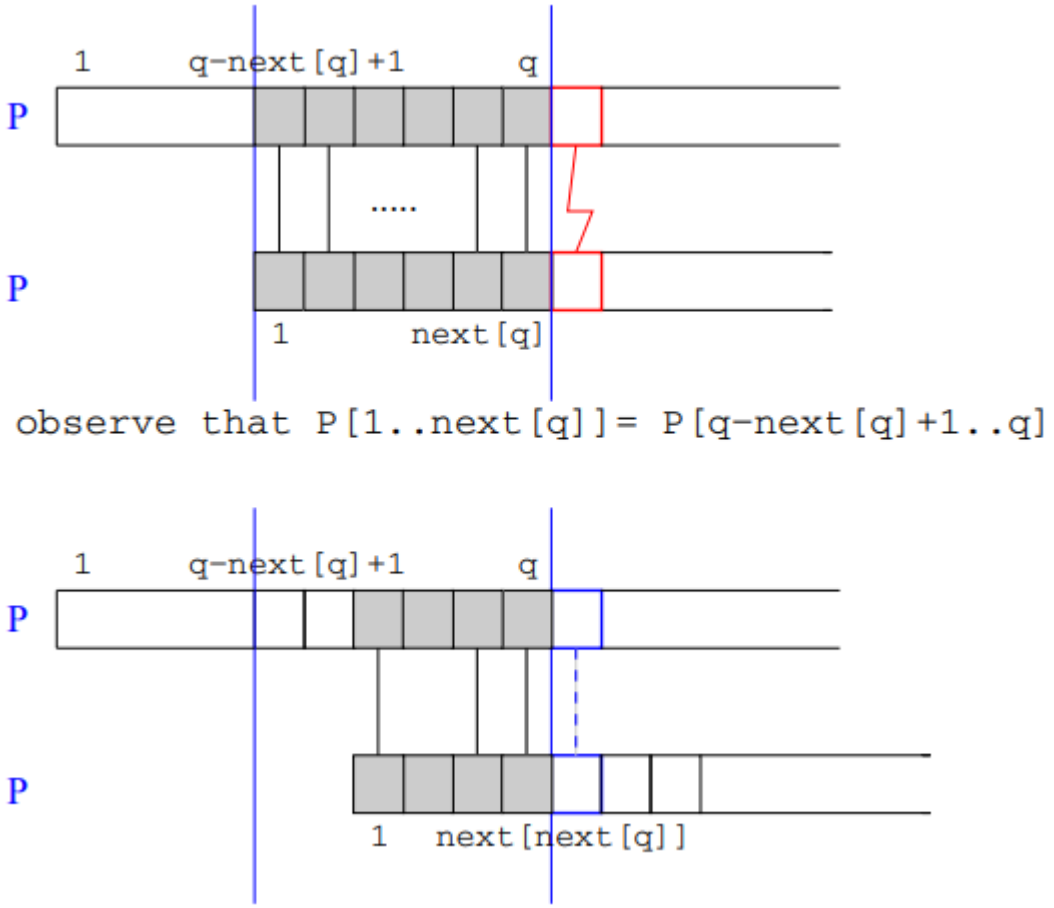
1. If `P[q+1]==P[next[q]+1]`,
   then `next[q+1]=next[q]+1`.



How to compute next[]/SP[] ?

2. If `P[q+1]!=P[next[q]+1]`, then do what?

   P should slide to a place such that the prefix of P `[1..next[q]]` occurs as a suffix of P `[q-next[q+1]..q]`; this information is stored in `next[next[q]]` !



KMP algorithm

```
KMP(T[ ], P[ ],n,m)
{
   preprocessing(P,SP,m);
   i  ← 1;   ▷ 본문 문자열 포인터
   j  ← -1;   ▷ 패턴 문자열 포인터
   ▷ n: 배열 T[ ]의 길이, m: 배열 P[ ]의 길이
   for (i=0;i<n;i++)
   {
        while (j >= 0 && T[i] != P[j+1]) j=SP[j];
        if (P[j+1]==T[i]) j++;
        if (j == m-1) {
                    A[i-m]에서 매치되었음을 알림;
                    j = SP [j];
        }
   }
}
```

✓수행시간: $\Theta(n)$

Preprocessing

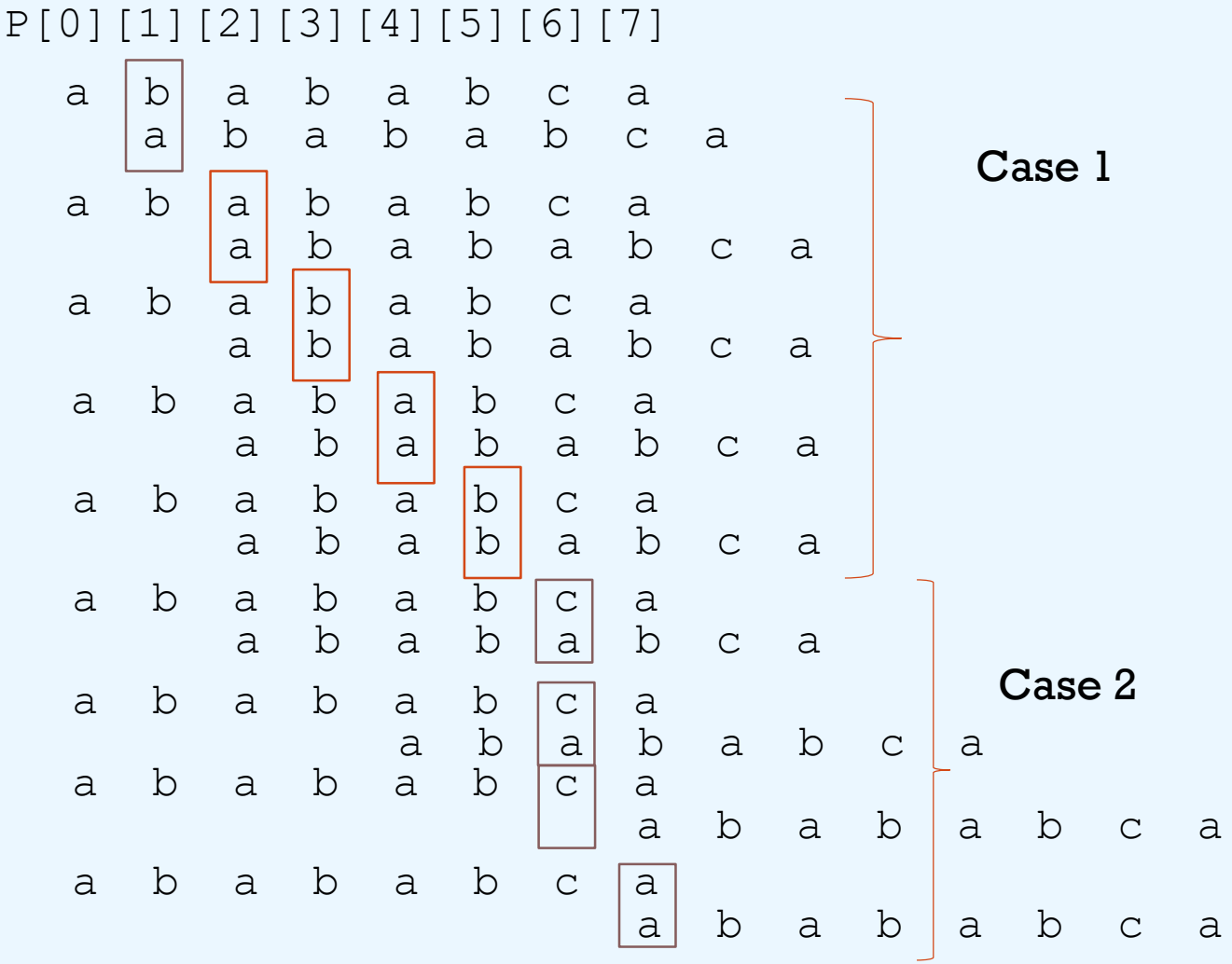
```
preprocessing(P,SP,m)
{
   SP [0]= -1;   k = -1;
   for (j=1;j<m;j++)
   {
       while (k >= 0 && P[j] != P[k+1]) k=SP[k];
       if (P[j] == P[k+1]) k++;
       SP [j]=k;
   }
}
```

✓수행시간: $\Theta(m)$

Example

- Next position
  - Failure function
  - Next function
  - preprocessing *SP* [];

i	next[i]
0	-1
1	-1
2	0
3	1
4	2
5	3
6	-1
7	0



Analysis of KMP

- O(m+n) 시간
  - 최대 접두부 테이블을 전처리 → O(m)
  - 매칭 → O(n)

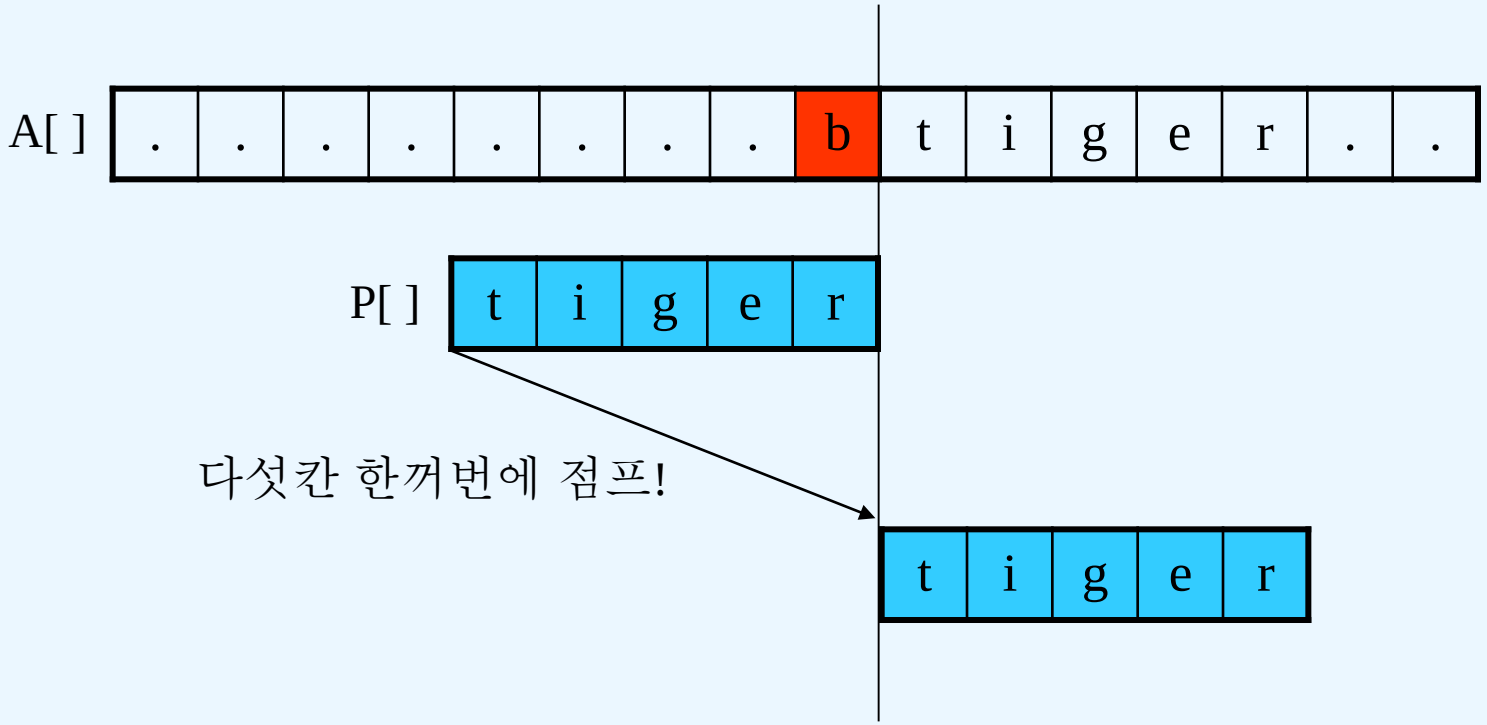
Boyer-Moore algorithm

Boyer-Moore Algorithm

- 앞의 매칭 알고리즘들의 공통점
  - 텍스트 문자열의 문자를 적어도 한번씩 훑는다
  - 따라서 최선의 경우에도 $\Omega(n)$
- 보이어-무어 알고리즘은 텍스트 문자를 다 보지 않아도 된다
  - 발상의 전환: 패턴의 오른쪽부터 비교한다

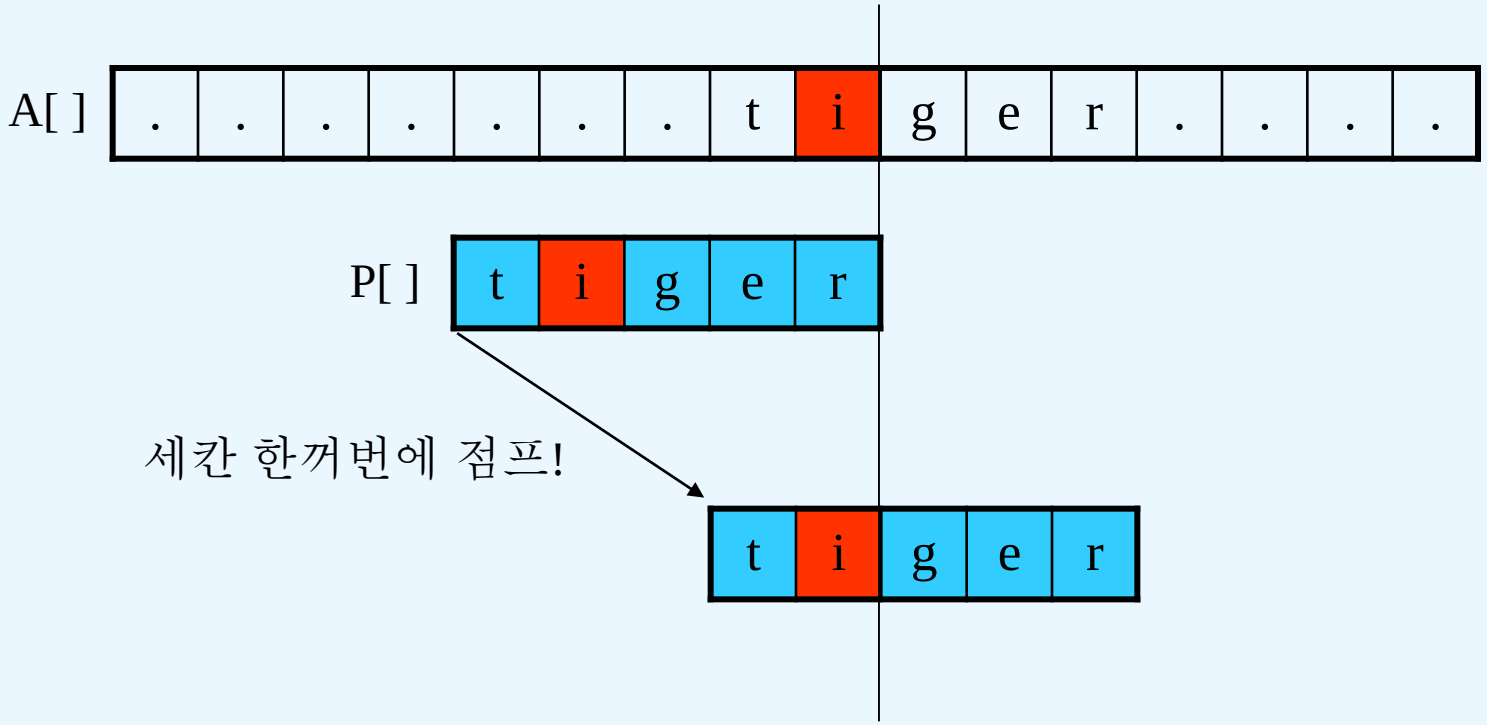
Motivation

상황: 텍스트의 b와 패턴의 r을 비교하여 실패했다



✓ 관찰: 패턴에 문자 b가 없으므로
패턴이 텍스트의 b를 통째로 뛰어넘을 수 있다

상황: 텍스트의 i와 패턴의 r을 비교하여 실패했다



✓ 관찰: 패턴에서 i가 r의 3번째 왼쪽에 나타나므로
패턴이 3칸을 통째로 움직일 수 있다

점프 정보 준비

패턴 “tiger”에 대한 점프 정보

오른쪽 끝문자	t	i	g	e	r	기타
jump	4	3	2	1	5	5

패턴 “rational”에 대한 점프 정보

오른쪽 끝문자	r	a	t	i	o	n	a	l	기타
jump	7	6	5	4	3	2	1	8	8

오른쪽 끝문자	r	t	i	o	n	a	l	기타
jump	7	5	4	3	2	1	8	8

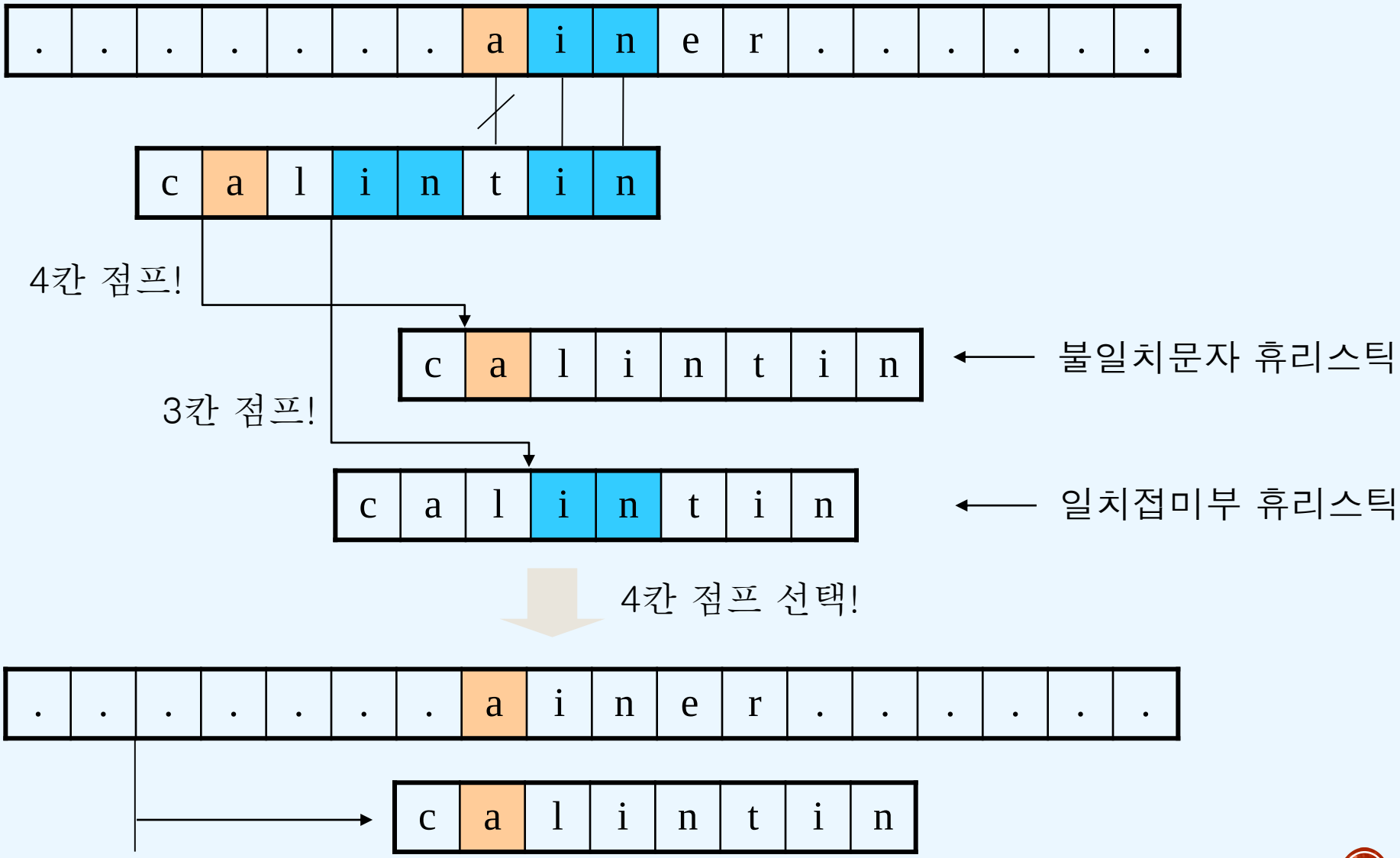
Boyer-Moore Algorithm

```
BoyerMoore (A[ ], P[ ])
{    ▷ n : 배열 A[ ]의 길이, m : 배열 P[ ]의 길이
   computeSkip(P, jump);
   i ← 1;
   while (i ≤ n − m+1) {
        j ← m;   k ← i + m −1;
        while ( j > 0 and P[j] = A[k]) {

                j--; k--;
        }
        if (j = 0) then A[i] 자리에서 매칭이 발견되었음을 알린다;
        i ← i + jump[A[i + m − 1]];
   }
}
```

- ✓최악의 경우 수행시간: $\Theta(mn)$
- ✓입력에 따라 다르지만 일반적으로 $\Theta(n)$보다 시간이 덜 든다

불일치문자 휴리스틱과 일치접미부 휴리스틱




. . . . a a l e r . . . . . .
r a t i o n a l
-1칸 점프!
r a t i o n a l
7칸 점프!
r a t i o n a l
불일치문자 휴리스틱
일치접미부 휴리스틱
7칸 점프 선택!
. . . . a a l e r . . . . . .
r a t i o n a l

복수 패턴 매칭

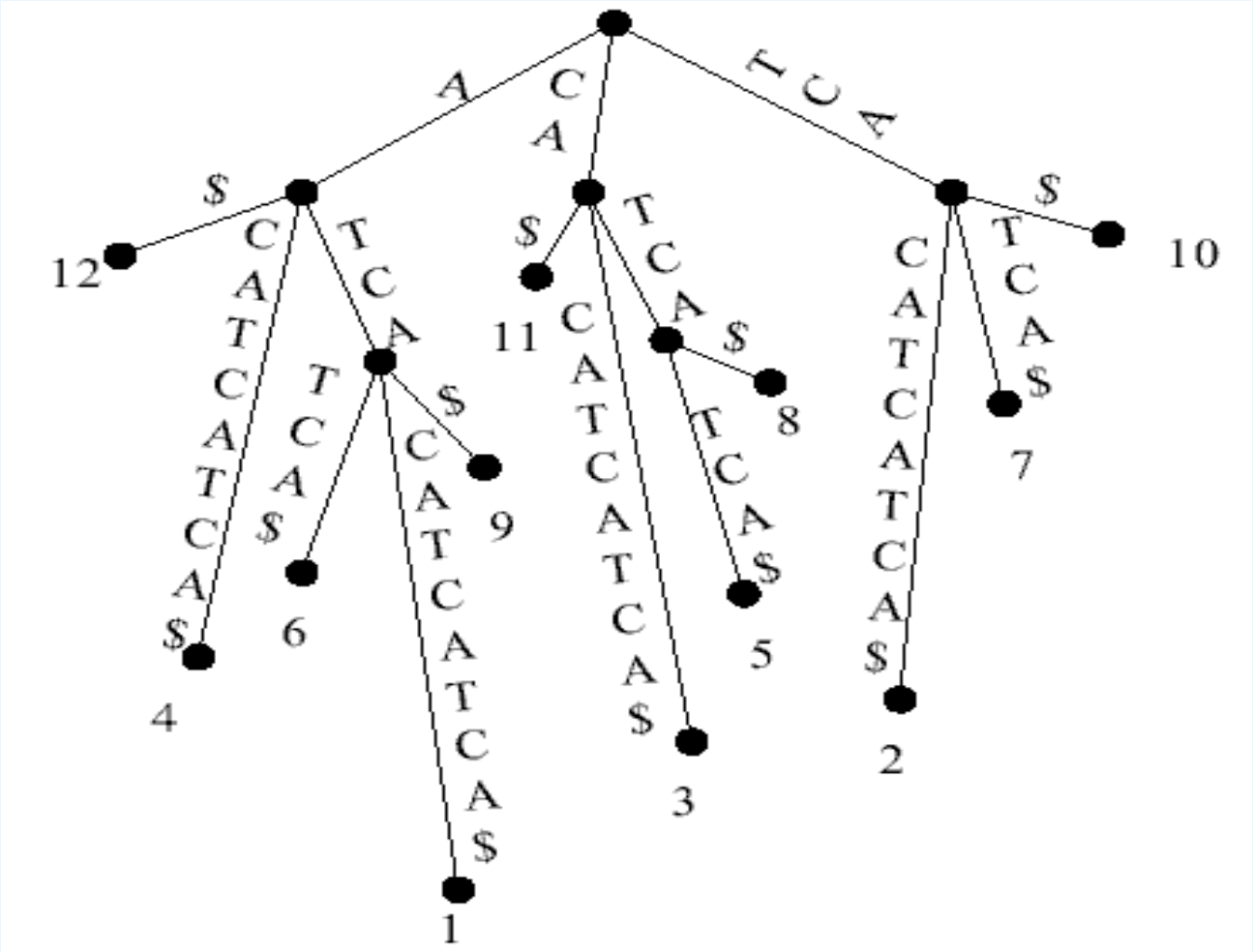
Suffixes

- 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	$S_{(12)}$

Suffixes

- 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

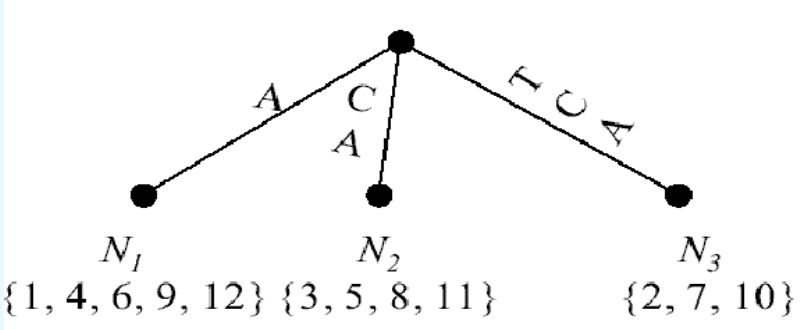
- S="ATCACATCATCA".
- Starting characters: "A", "C", "T"
- In N_3,

  $S(2)$ ="TCACATCATCA"

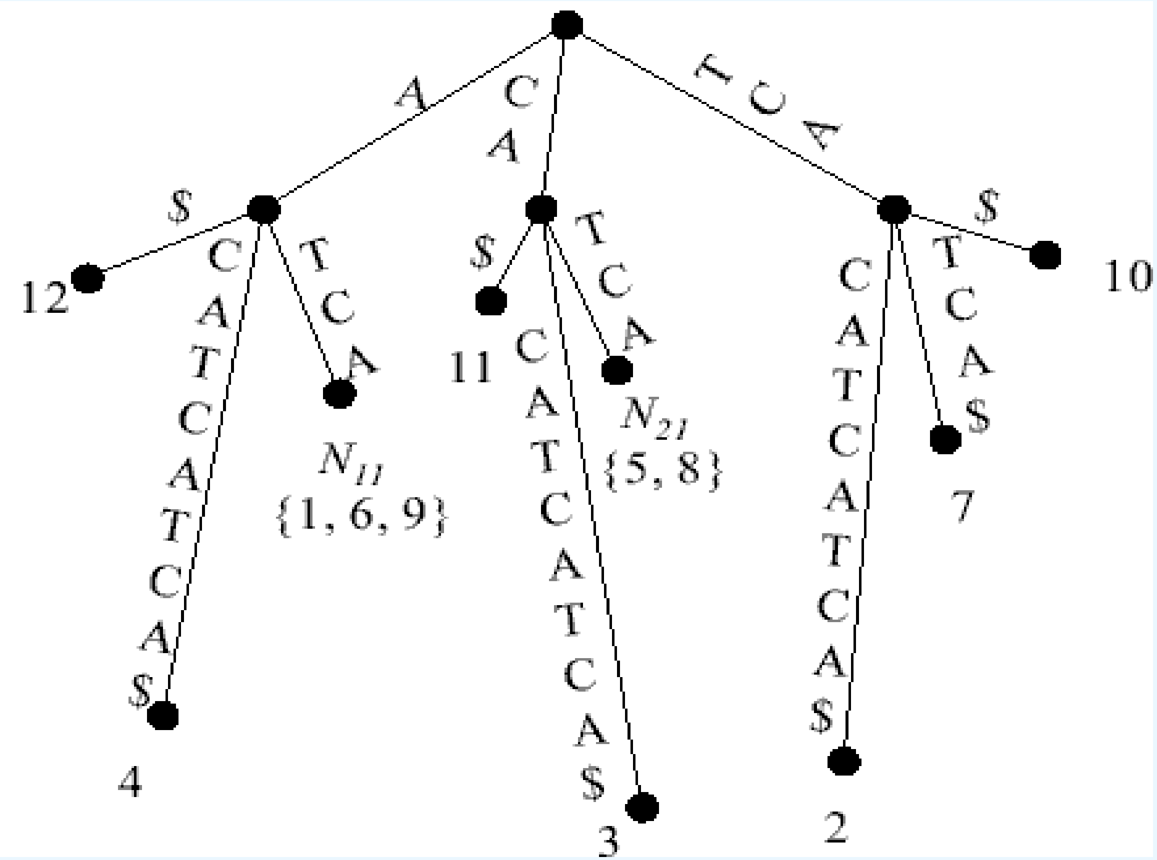
  $S(7)$ ="TCATCA"

  $S(10)$ ="TCA"
- Longest common prefix of N_3 is "TCA"

S(1):ATCACATCATCA

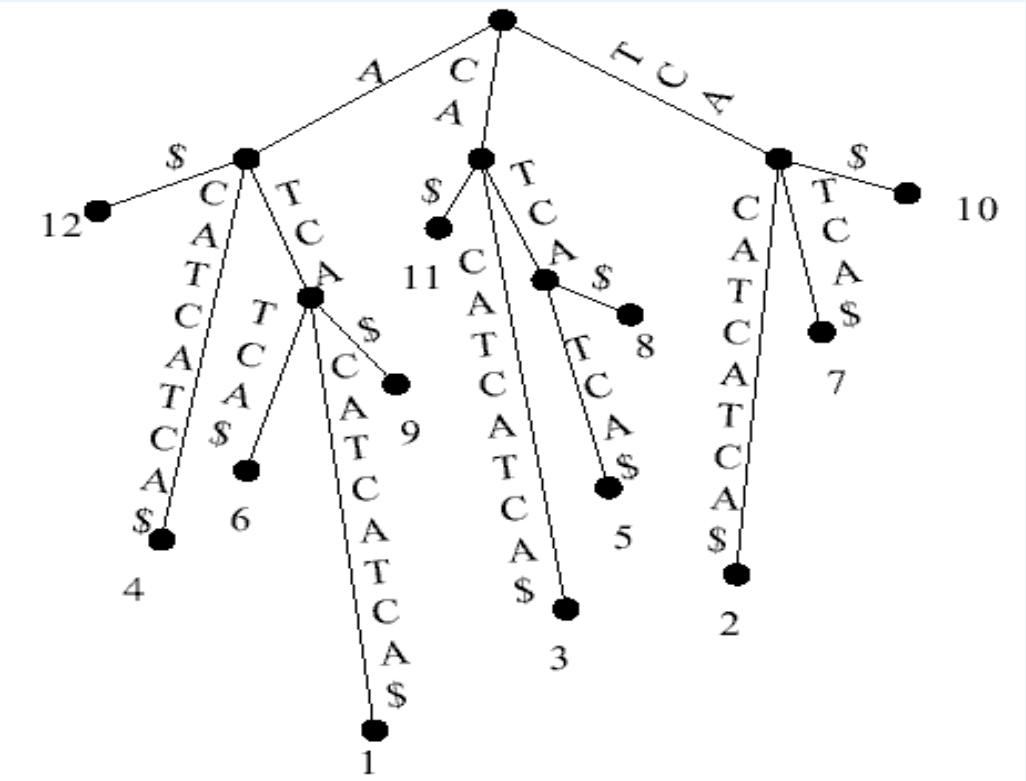


- S=“ATCACATCATCA”.
- Second recursion:



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 comp licated algorithm).
- To search a pattern P of length m on a suffix tree needs O(m) comparisons.
- Exact string matching: O(n+m) time