String Search (Matching) Algorithms

This lecture is based on the slides of 한빛미디어㈜

Index

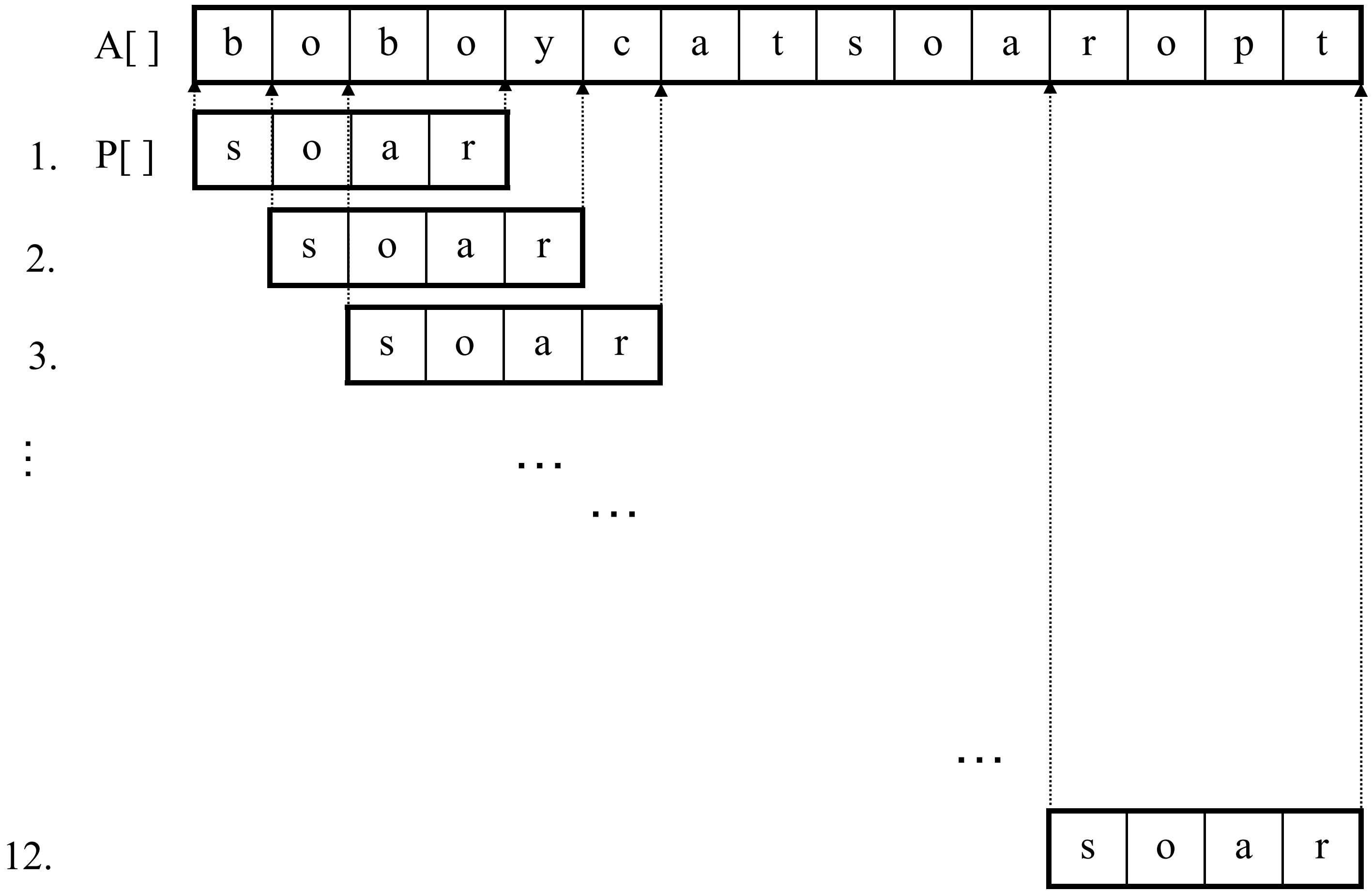
- Brute-Force Search
- Rabin-Karp
- Automata
- KMP
- Boyer-Moore

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

Brute-Force Matching Algorithm

원시적인 매칭의 작동 원리

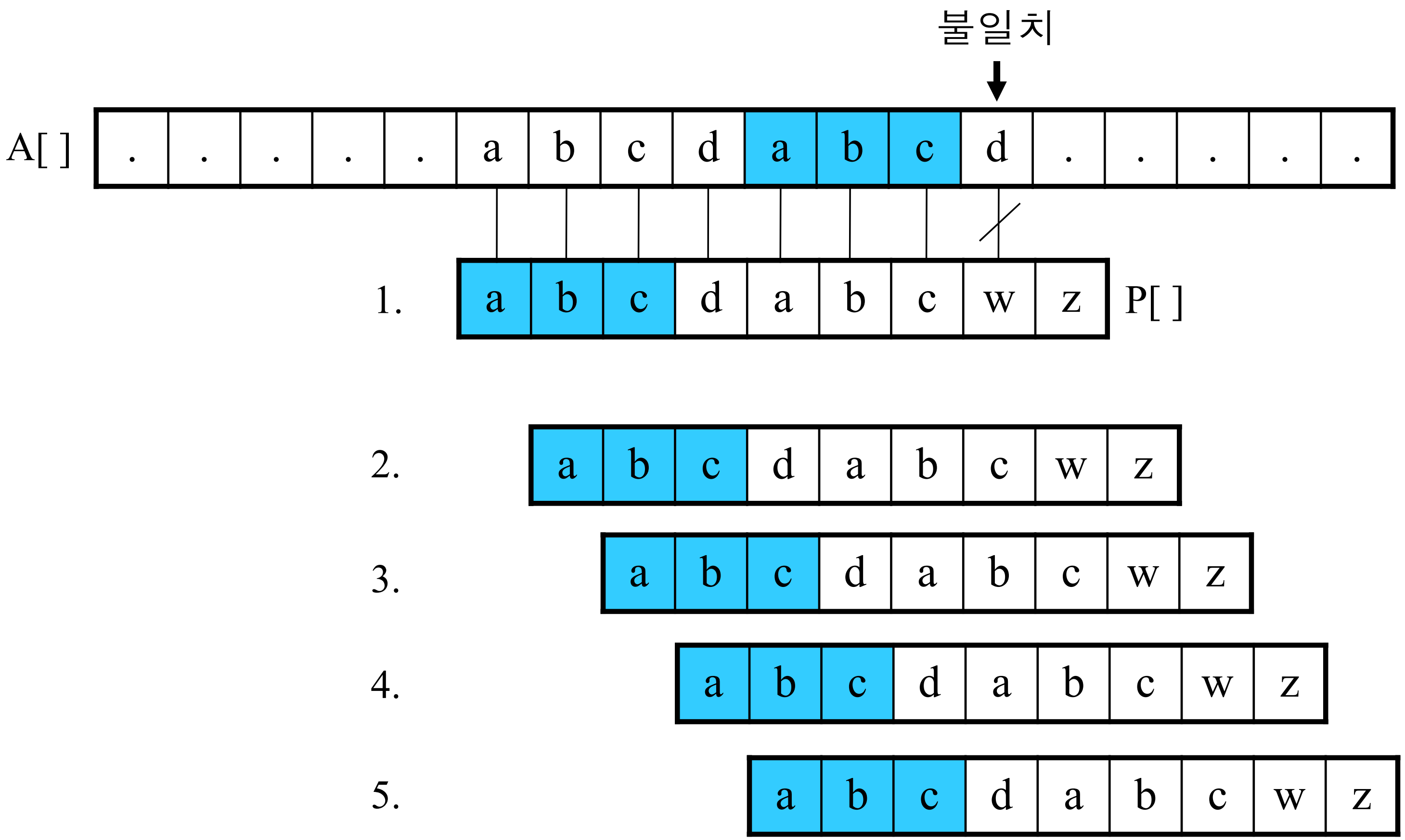


원시적인 매칭

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

✓ 수행시간: $O(mn)$

원시적인 매칭이 비효율적인 예



Rabin-Karp Algorithm

라빈-카프Rabin-Karp 알고리즘

- 문자열 패턴을 수치로 바꾸어 문자열의 비교를 수치 비교로 대신한다
- 수치화
  - 가능한 문자 집합 Σ의 크기에 따라 진수가 결정된다
  - 예: $\Sigma = \{a, b, c, d, e\}$
    - $|\Sigma| = 5$
    - 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
  - $a_i = A[i+m-1] + d(A[i+m-2] + d(A[i+m-3] + d(\ldots + d(A[i]))\ldots)$
  - $\Theta(m)$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
  - $a_i = d(a_{i-1} - d^{m-1}A[i-1]) + A[i+m-1]$
  - d^{m-1}은 반복 사용되므로 미리 한번만 계산해 두면 된다
  - 곱셈 2회, 덧셈 2회로 충분

수치화를 이용한 매칭의 예

P[ ]

e	e	a	a	b

$p = 4*5^4 + 4*5^3 + 0*5^2 + 0*5^1 + 1 = 3001$

A[ ]

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

$a_1=0*5^4+2*5^3+4*5^2+1*5^1+1 = 356$

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

$a_2= 5(a_1-0*5^4)+2 = 1782$

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

$a_3=5(a_2-2*5^4)+4 = 2664$

...

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

$a_7=5(a_6-2*5^4)+1 = 3001$

...

수치화를 이용해 매칭을 체크하는 알고리즘

```
basicRabinKarp(A, P, d, q)
{
    ▷ n : 배열 A[ ]의 길이,  m : 배열 P[ ]의 길이
    p ← 0; a_1 ← 0;
    for i ← 1 to m {            ▷ a_1 계산
        p ← dp + P[i];
        a_1 ← da_1 + A[i];
    }
    for i ← 1 to n-m+1 {
        if (i ≠ 1) then a_i ← d(a_{i-1} – d^{m-1}A[i-1]) + A[i+m-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
  - $a_i = d(a_{i-1} - d^{m-1}A[i-1]) + A[i+m-1]$ 대신
    $b_i = (d(b_{i-1} - (d^{m-1} \bmod q) A[i-1]) + A[i+m-1]) \bmod q$ 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

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

...

라빈-카프 알고리즘

```
RabinKarp(A, P, d, q)
{
    ▷ n : 배열 A[ ]의 길이,  m : 배열 P[ ]의 길이
    p ← 0; b_1 ← 0;
    for i ← 1 to m {                                        ▷ b_1 계산
        p ← (dp + P[i]) mod q;
        b_1 ← (db_1 + A[i]) mod q;
    }
    h ← d^(m-1) mod q;
    for i ← 1 to n-m+1 {
        if (i ≠ 1) then b_i ← (d(b_(i-1) – hA[i-1]) + A[i+m-1]) mod q;
        if (p = b_i) then
            if (P[1…m] = A[i…i+m-1]) then
                A[i] 자리에서 매칭이 되었음을 알린다;
    }
}
```

✓ 평균 수행시간: $\Theta(n)$

라빈-카프 알고리즘

What if many characters can be mapped to a single code?
-> slow down, hash collision

Automata for Matching Algorithm

ababaca를 체크하는 오토마타

S: dvganbbactababaababacababacaagbk…
ababaca
 ababaca

S: dvganbbactababaababacababacaagbk…
 ababaca
  ababaca

S: dvganbbactababaababacababacaagbk…
ababaca
 ababaca

S: dvganbbactababaababacababacaagbk…
ababaca
  ababaca

ababaca를 체크하는 오토마타

S: dvganbbactababaababacababababacaagbk…

ababaca

S: dvganbbactababaababacababababacaagbk…

ababaca

ababaca

How do we know the shift length?

ababaca를 체크하는 오토마타

S: dvganbbactababaababacababababacaagbk…

ababaca

S: dvganbbactababaababacababababacaagbk…

ababaca

S: dvganbbactababaababacababababacaagbk…

ababaca

S: dvganbbactababaababacababababacaagbk…

ababaca

ababaca를 체크하는 오토마타

To skip pattern matching at an index, we need to be sure that the substring starting at the index is not matching to the prefix of the pattern.

S: dvganbbactababaababacababab acaagbk…

ababaca

If the substring is matching to a prefix of the pattern, then we need to check the matching of the subsequent sequence of the substring.

S: dvganbbactababaababacababab acaagbk…

ababaca

ababaca를 체크하는 오토마타

N-1 shift is possible. N: the number of matching characters

S: dvganbbactababaababacababababacaagbk…

ababaca

If there are many matching cases, we need to evaluate the largest N first.
If not, we will miss a matching case.

S: dvganbbactababaababacababababacaagbk…

ababaca

ababaca

missed, if we skip too many characters

ababaca를 체크하는 오토마타

What to check

S1 : A sequence P_k + new alphabet (failed to match to the pattern)

S2 : the pattern

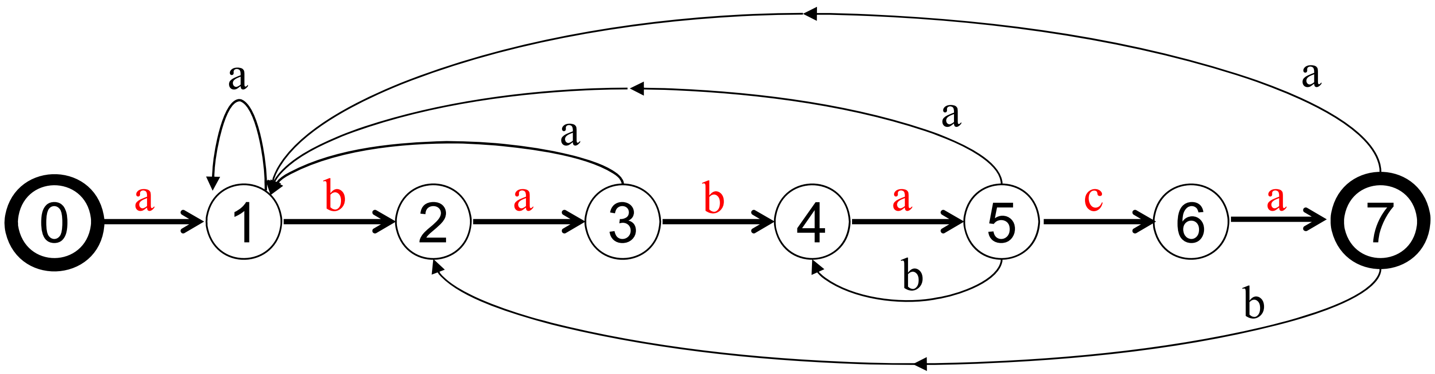
S3 = { sequence | sequence = A suffix of S1 matching to the prefix of S2}

We can not skip matching for any sequence in S3.
The largest sequence in S3 should be the first location to start new matching.

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

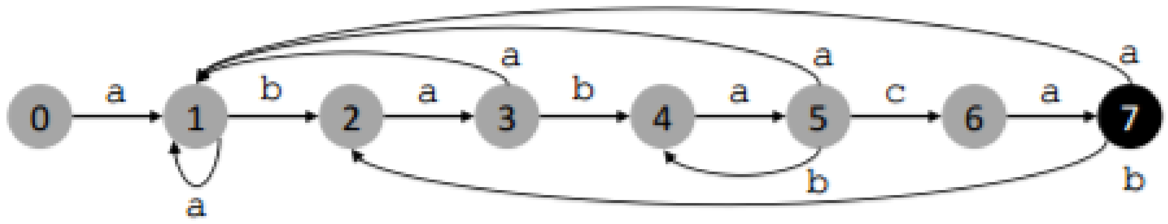
상태 \ 입력문자	a	b	c	기타
0	1	0	0	0
1	1	2	0	0
2	3	0	0	0
3	1	4	0	0
4	5	0	0	0
5	1	4	6	0
6	7	0	0	0
7	1	2	0	0

String-Matching Automata

- For every pattern $P[1..m]$, we need to construct a string-matching automaton in preprocessing
  - the state set Q is $\{0,1,\dots,m\}$, where start state q_0 is state 0 and state m is the only accepting state
  - the transition function is defined as $\delta(q,a) = \sigma(P_q a)$ for any state q and character a
- Suffix function σ for a given pattern $P[1..m]$
  - $\sigma: \Sigma \to \{0,1,\dots,m\}$ such that $\sigma(x) = \max\{k: P_k \sqsupset x\}$ is the length of the longest prefix of P that is a suffix of x
  - for a pattern P of length m, $\sigma(x) = m$ if and only if $P \sqsupset x$
  - if $x \sqsupset y$, then $\sigma(x) \leq \sigma(y)$

Example

- Assume pattern $P = \mathtt{ababaca}$



  - 8 states and a "spine" of forward transitions
  - $\delta(1, \mathtt{a}) = 1$, since $P_1\mathtt{a} = \mathtt{a}\mathbf{a}$ and $\sigma(P_1\mathtt{a}) = 1$
  - $\delta(3, \mathtt{a}) = 1$, since $P_3\mathtt{a} = \mathtt{aba}\mathbf{a}$ and $\sigma(P_3\mathtt{a}) = 1$
  - $\delta(5, \mathtt{a}) = 1$ since $P_5\mathtt{a} = \mathtt{ababa}\mathbf{a}$ and $\sigma(P_5\mathtt{a}) = 1$
  - $\delta(5, \mathtt{b}) = 4$, since $P_5\mathtt{b} = \mathtt{ab}\mathbf{abab}$ and $\sigma(P_5\mathtt{b}) = 4$
  - $\delta(7, \mathtt{a}) = 1$, since $P_7\mathtt{a} = \mathtt{ababaca}\mathbf{a}$ and $\sigma(P_7\mathtt{a}) = 1$
  - $\delta(7, \mathtt{b}) = 2$, since $P_7\mathtt{b} = \mathtt{ababac}\mathbf{ab}$ and $\sigma(P_7\mathtt{b}) = 2$

Computing the Transition Function δ

```
COMPUTE-TRANSITION-FUNCTION(P, Σ)
m ← length[P]
for q ← 0 to m
    do for each character a ∈ Σ
           do k ← min(m + 1, q + 2)
              repeat k ← k − 1
                until P_k ⊐ P_q a
              δ(q, a) ← k
return δ
```

- Computing transition function takes time $O(m^3|\Sigma|)$
  - outer two **for** loops contribute a factor of $m^3|\Sigma|$
  - inner **repeat** loop can run at most $m + 1$ times
  - test $P_k \sqsupset P_q a$ can require up to m comparisons

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;
    }
}
```

✓ 총 수행시간: $\Theta(n)$

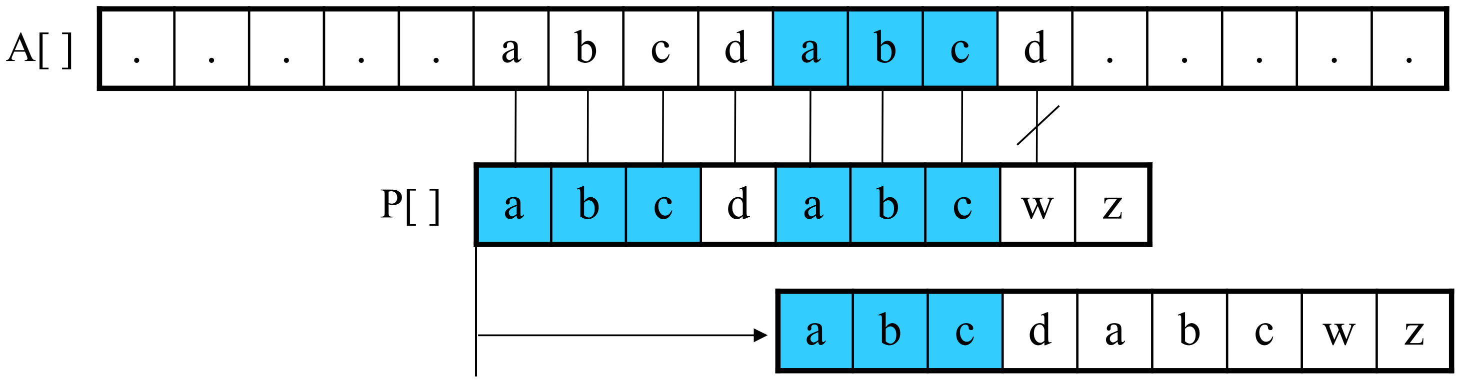
전처리 포함 총 수행시간: $\Theta(n + |\Sigma|m^3)$

KMP Algorithm

- 오토마타 구성 시간을 줄이면 $\Theta(n + |\sum|m)$ 까지도 가능..
- 전처리 효율 향상 -> KMP algorithm

KMP Knuth-Morris-Pratt 알고리즘

- 오토마타를 이용한 매칭과 동기가 유사
- 공통점
  - 매칭에 실패했을 때 돌아갈 상태를 준비해둔다
  - 오토마타를 이용한 매칭보다 준비 작업이 단순하다



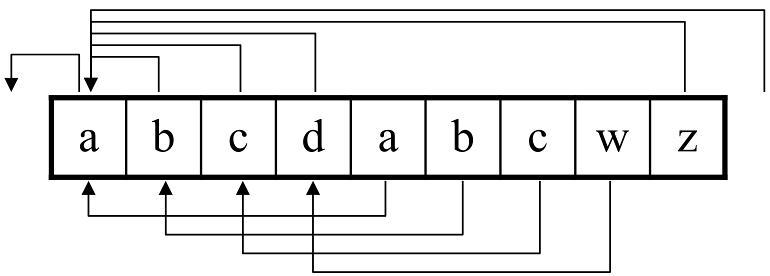
매칭이 실패했을 때 돌아갈 곳 준비 작업

	1	2	3	4	5	6	7	8	9
P[ ]	a	b	c	d	a	b	c	w	z

$\pi[8] = 4$

텍스트에서 abcdabc까지는 매치되고, w에서 실패한 상황
패턴의 맨앞의 abc와 실패 직전의 abc는 동일함을 이용할 수 있다
실패한 텍스트 문자와 P[4]를 비교한다

	1	2	3	4	5	6	7	8	9	10
π[ ]	0	1	1	1	1	2	3	4	1	1



패턴의 각 위치에 대해
매칭에 실패했을 때
돌아갈 곳을 준비해 둔다

Failure Table

A table to remember which state to go
when matching is failed.
Simple version of DFA.

i	0	1	2	3	4	5	6
$W[i]$	A	B	C	D	A	B	D
$T[i]$	-1	0	0	0	0	1	2

```
algorithm kmp_search:
  input:
    an array of characters, S (the text to be searched)
    an array of characters, W (the word sought)
  output:
    an integer (the zero-based position in S at which W is found)

  define variables:
    an integer, m ← 0 (the beginning of the current match in S)
    an integer, i ← 0 (the position of the current character in W)
    an array of integers, T (the table, computed elsewhere)
```

```
while m + i < length(S) do
    if W[i] = S[m + i] then
        if i = length(W) - 1 then
            return m
        let i ← i + 1
    else
        if T[i] > -1 then
            let m ← m + i - T[i], i ← T[i]
        else
            let m ← m + 1, i ← 0

(if we reach here, we have searched all of S unsuccessfully)
return the length of S
```

✓수행시간: $\Theta(n)$

```
algorithm kmp_table:
    input:
        an array of characters, W (the word to be analyzed)
        an array of integers, T (the table to be filled)
    output:
        nothing (but during operation, it populates the table)

    define variables:
        an integer, pos ← 2 (the current position we are computing in T)
        an integer, cnd ← 0 (the zero-based index in W of the next
character of the current candidate substring)

    (the first few values are fixed but different from what the algorithm
might suggest)
    let T[0] ← -1, T[1] ← 0
```

```
while pos < length(W) do
    (first case: the substring continues)
    if W[pos-1] = W[cnd] then
        let T[pos] ← cnd + 1, cnd ← cnd + 1, pos ← pos + 1

    (second case: it doesn't, but we can fall back)
    else if cnd > 0 then
        let cnd ← T[cnd], T[pos] ← 0

    (third case: we have run out of candidates.  Note cnd = 0)
    else
        let T[pos] ← 0, pos ← pos + 1
```

i	0	1	2	3	4	5	6
$W[i]$	A	B	C	D	A	B	D
$T[i]$	-1	0	0	0	0	1	2

✓수행시간: $\Theta(m)$

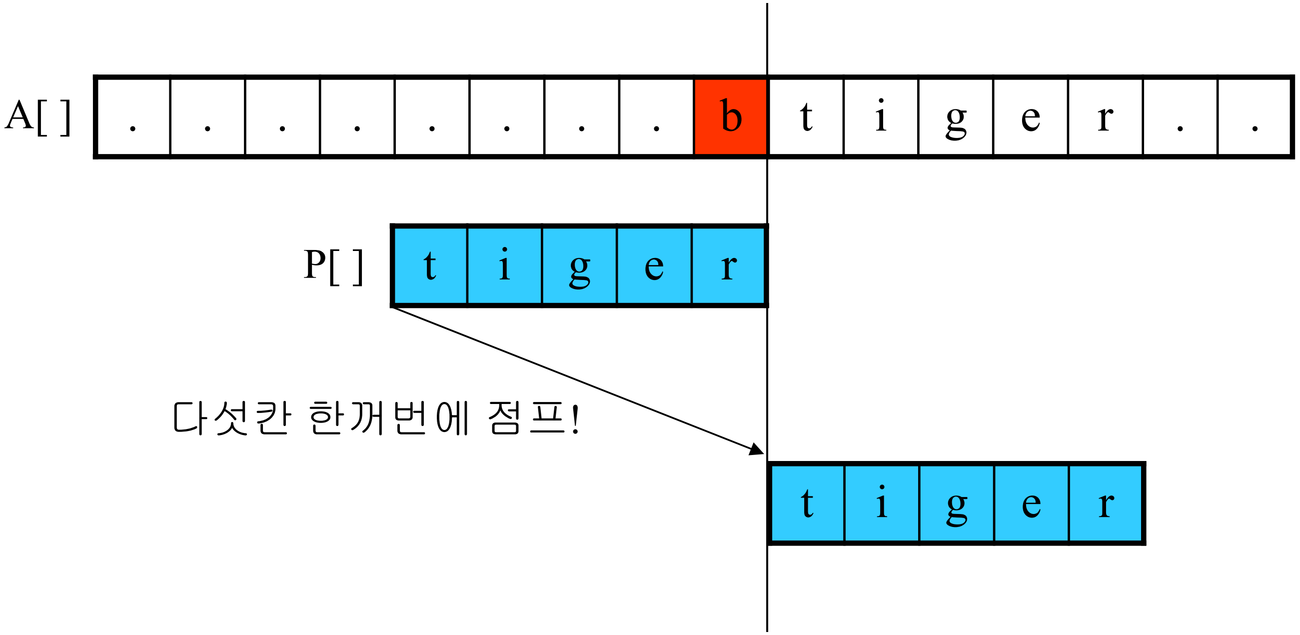
Boyer-Moore Algorithm

보이어-무어Boyer-Moore 알고리즘

- 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

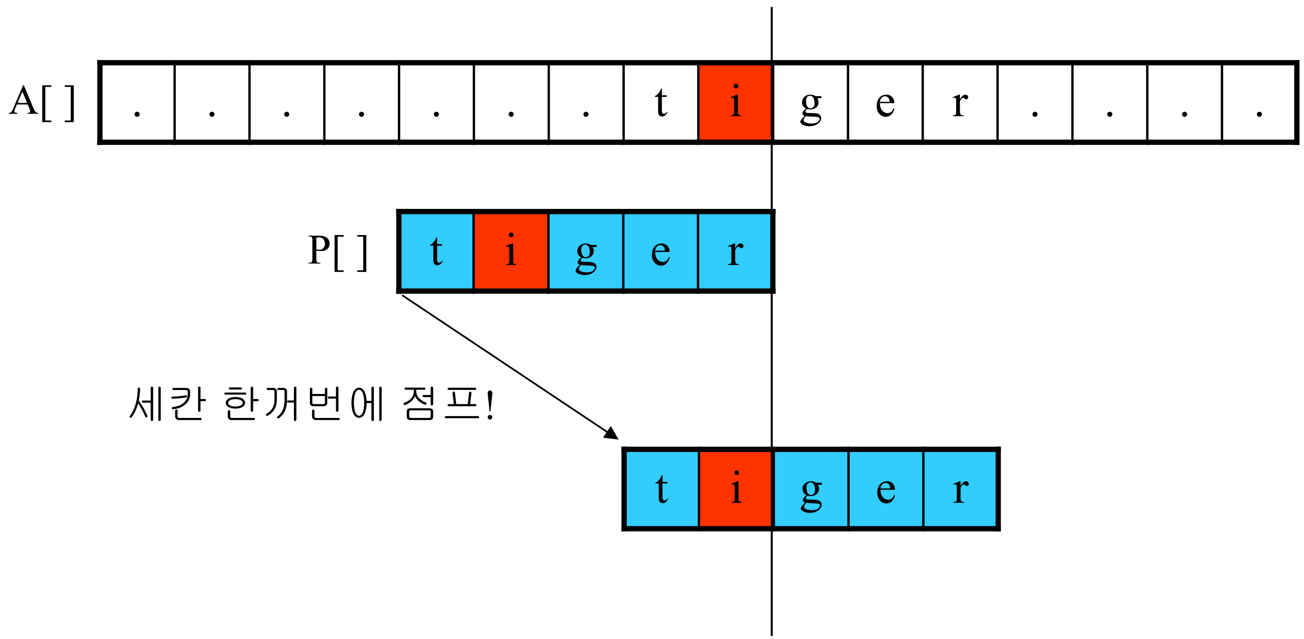
Motivation – Bad Character Shift

상황: 텍스트의 b와 패턴의 r을 비교하여 실패했다



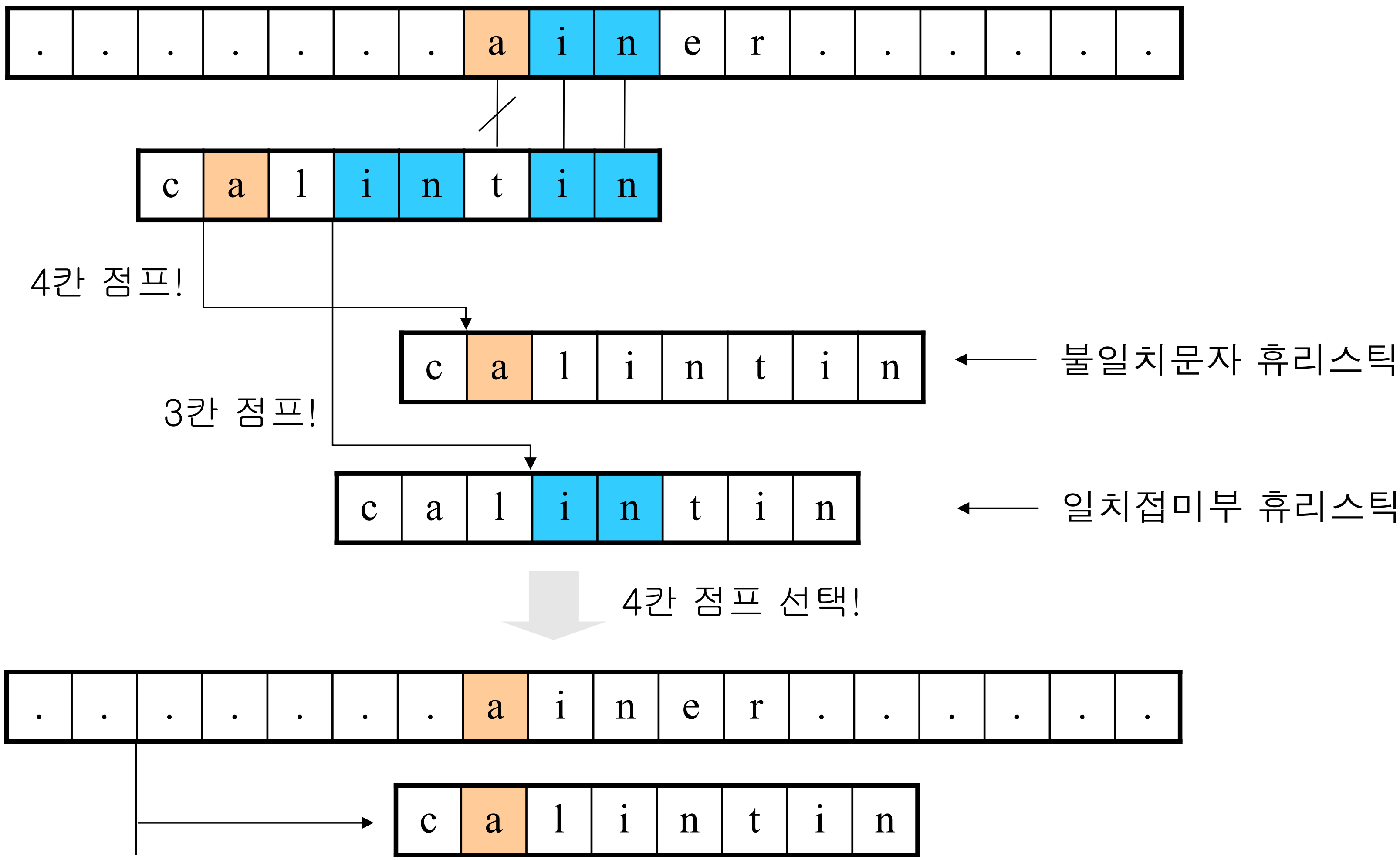
✓ 관찰: 패턴에 문자 b가 없으므로
패턴이 텍스트의 b를 통째로 뛰어넘을 수 있다

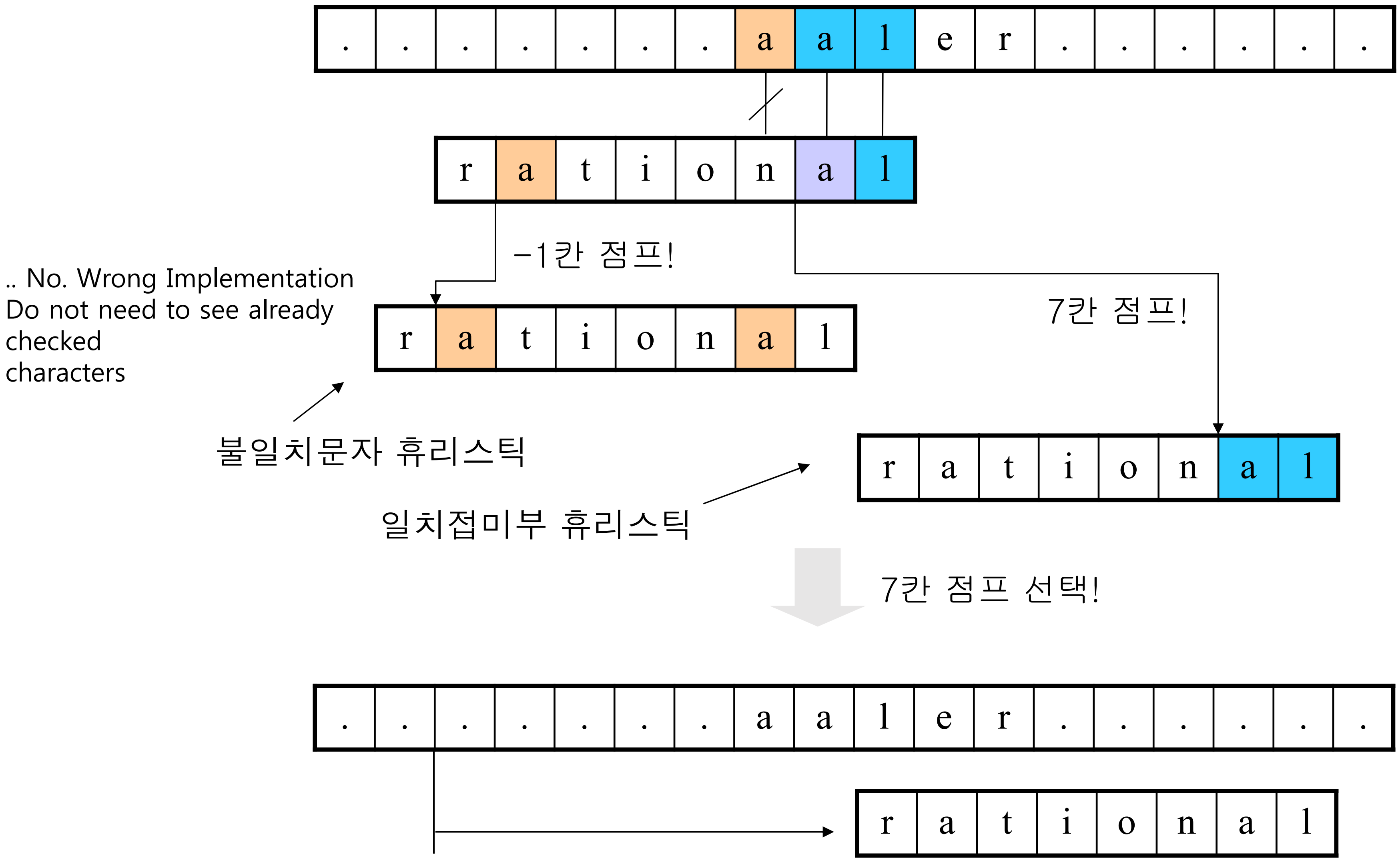
상황: 텍스트의 i와 패턴의 r을 비교하여 실패했다



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

Bad Character Shift vs Good Suffix Shift




. . . . a a l e r . . . . . .
r a t i o n a l
-1칸 점프!
.. No. Wrong Implementation
Do not need to see already
checked
characters
r a t i o n a l
7칸 점프!
불일치문자 휴리스틱
r a t i o n a l
일치접미부 휴리스틱
7칸 점프 선택!
. . . . a a l e r . . . . . .
r a t i o n a l

보이어-무어-호스풀 알고리즘

Simpler version of Boyer-Moore algorithm (Boyer-Moore algorithm is more complex… we didn't cover this algorithm)

Bad Character Matching (x) -> matching the last character we searched

Only Bad Character Matching

```
BoyerMooreHorspool(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;  // j: index of P, k : index of A
        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

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

축약: 어휘별 Unique 이동거리 책정, 같은 어휘 존재하는 경우 최소거리로 설정 (안전) (right most occurrence)

보이어-무어-호스풀 알고리즘

Simpler version of Boyer-Moore algorithm
Bad Character Matching (x) -> matching the last character we searched

BoyerMooreHorspool(A[ ], P[ ])
{ ▷ n : 배열 A[ ]의 길이, m : 배열 P[ ]의 길이
  computeSkip(P, *jump*);
  $i \leftarrow 1$;
  while ($i \le n - m+1$) {
    $j \leftarrow m$; $k \leftarrow i + m - 1$; // j: index of P, k : index of A
    while ($j > 0$ **and** P[j] = A[k]) {
      j--; k--;
    }
    if ($j = 0$) **then** A[i] 자리에서 매칭이 발견되었음을 알린다;
    $i \leftarrow i + jump[A[i + m - 1]]$;
  }
}

✓최악의 경우 수행시간: $\Theta(mn)$

✓입력에 따라 다르지만 일반적으로 $\Theta(n)$보다 시간이 덜 든다