Ch.12 문자열 매칭(탐색)

학습목표

- ✓ 원시적인 매칭 방법에 깃든 비효율성을 감지할 수 있도록 한다.
- ✓ 오토마타를 이용한 매칭 방법을 이해한다.
- ✓ 라빈 - 카프 알고리즘의 수치화 과정을 이해한다.
- ✓ KMP 알고리즘을 이해하고, 오토마타를 이용한 방법과 비교해 이점을 이해하도록 한다.
- ✓ 보이어 - 무어 알고리즘의 개요를 이해하고, 다른 매칭 알고리즘들에 비해 어떤 특장점이 있는지 이해한다.

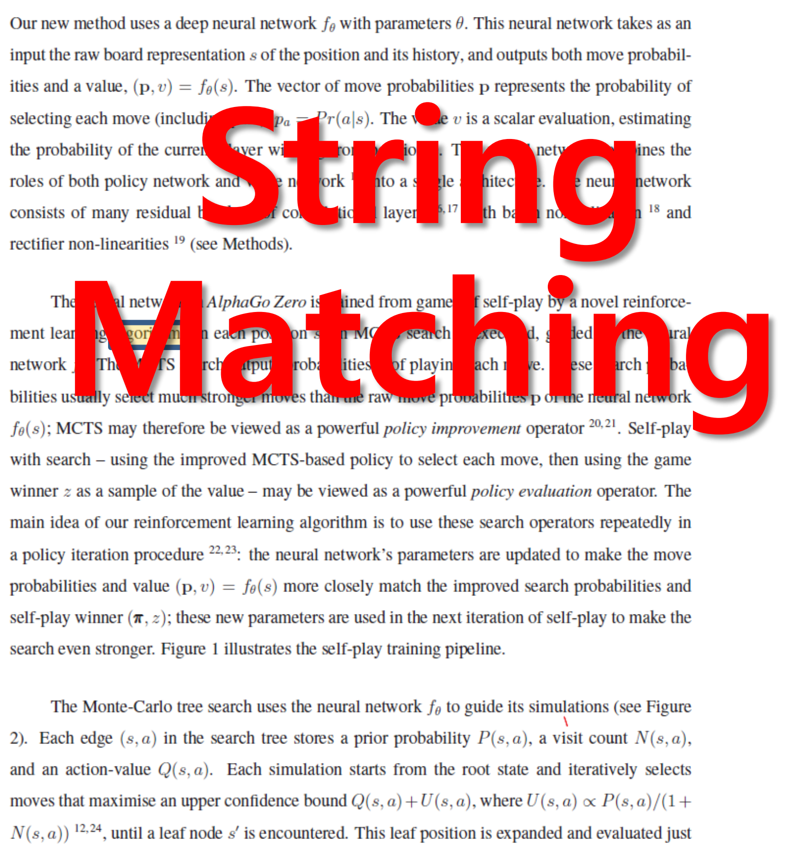
문자열 매칭

■ 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


Our new method uses a deep neural network f_θ with parameters θ. This neural network takes as an input the raw board representation s of the position and its history, and outputs both move probabilities and a value, $(\mathbf{p}, v) = f_\theta(s)$. The vector of move probabilities $\mathbf{p}$ represents the probability of selecting each move (includi [illegible] $p_a = Pr(a|s)$. The v [illegible] e v is a scalar evaluation, estimating the probability of the curre [illegible] ayer wi [illegible]. T [illegible] netw [illegible] ines the roles of both policy network and v [illegible] e n [illegible] ork [illegible] nto a s [illegible] gle [illegible] hitec [illegible] e. [illegible] e neur [illegible] network consists of many residual b [illegible] co [illegible] tio [illegible] layer [illegible] th ba [illegible] no [illegible] n [18] and rectifier non-linearities [19] (see Methods).

The [illegible] al netw [illegible] *AlphaGo Zero* is [illegible] ined from game [illegible] f self-play by a novel reinforcement lear [illegible] each po [illegible] on [illegible] MC [illegible] search [illegible] xec [illegible] d, g [illegible] ded [illegible] the [illegible] ral network [illegible] The [illegible] TS [illegible] rch [illegible] utput [illegible] robab [illegible] ities [illegible] of playin [illegible] ach m [illegible] ve. [illegible] ese [illegible] rch [illegible] ba bilities usually select much stronger moves than the raw move probabilities p of the neural network $f_\theta(s)$; MCTS may therefore be viewed as a powerful *policy improvement* operator [20,21]. Self-play with search – using the improved MCTS-based policy to select each move, then using the game winner z as a sample of the value – may be viewed as a powerful *policy evaluation* operator. The main idea of our reinforcement learning algorithm is to use these search operators repeatedly in a policy iteration procedure [22,23]: the neural network's parameters are updated to make the move probabilities and value $(\mathbf{p}, v) = f_\theta(s)$ more closely match the improved search probabilities and self-play winner $(\boldsymbol{\pi}, z)$; these new parameters are used in the next iteration of self-play to make the search even stronger. Figure 1 illustrates the self-play training pipeline.

The Monte-Carlo tree search uses the neural network f_θ to guide its simulations (see Figure 2). Each edge (s, a) in the search tree stores a prior probability $P(s, a)$, a visit count $N(s, a)$, and an action-value $Q(s, a)$. Each simulation starts from the root state and iteratively selects moves that maximise an upper confidence bound $Q(s, a) + U(s, a)$, where $U(s, a) \propto P(s, a)/(1 + N(s, a))$ [12,24], until a leaf node s' is encountered. This leaf position is expanded and evaluated just

원시적인 매칭의 작동

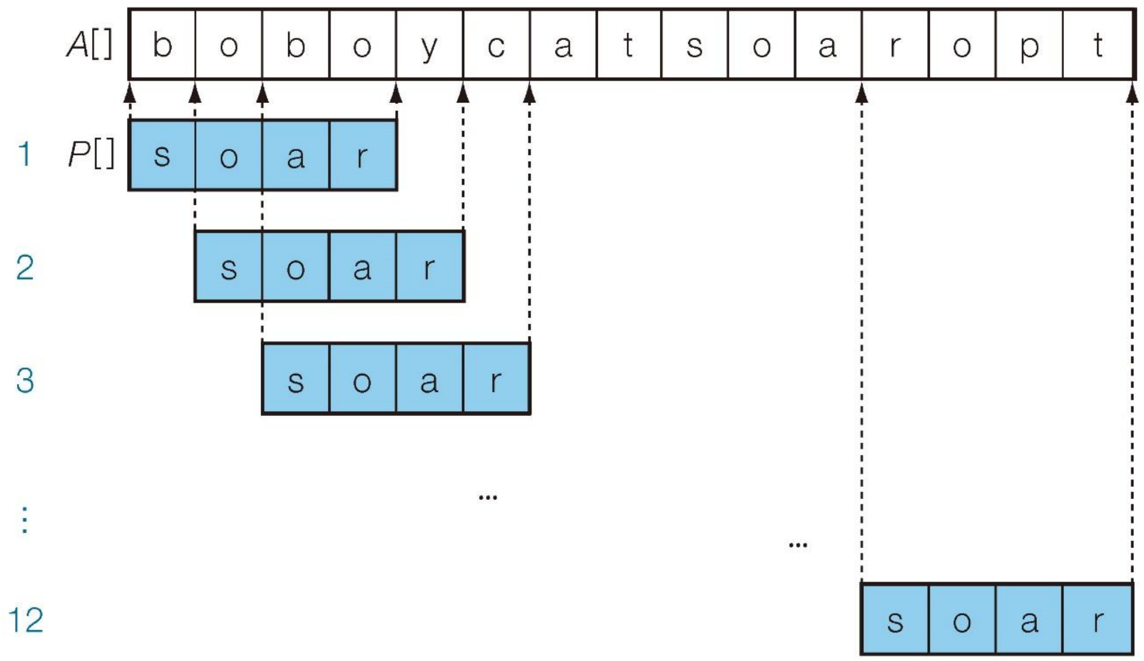


그림 12-1 원시적 매칭 방법

원시적인 매칭의 비효율적인 예

알고리즘 12-1 **원시적 매칭**

naiveMatching(A[], P[]):

▷ n: 배열 A[]의 길이, m: 배열 P[]의 길이

❶ for $i \leftarrow 1$ to $n-m+1$

❷ if $(P[1...m] = A[i...i+m-1])$

$A[i]$ 자리에서 매칭이 발견되었음을 알린다.

✓ 수행 시간: $O(mn)$

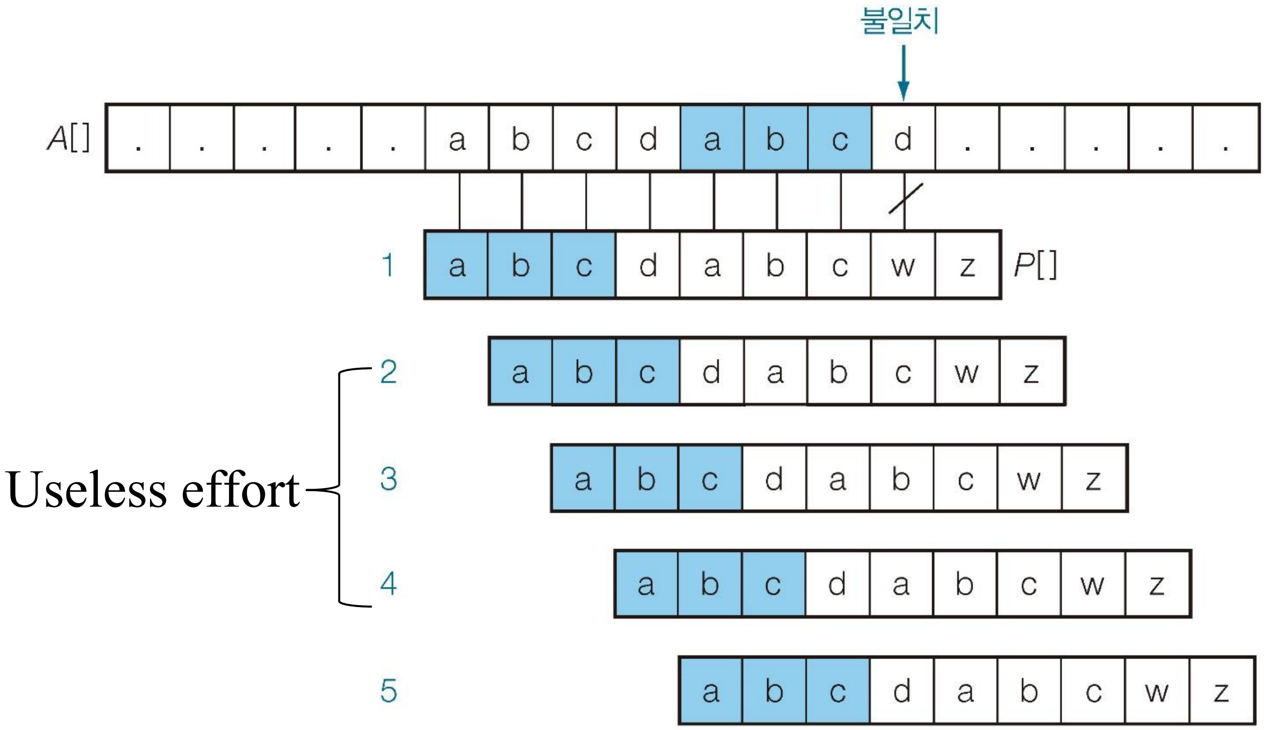
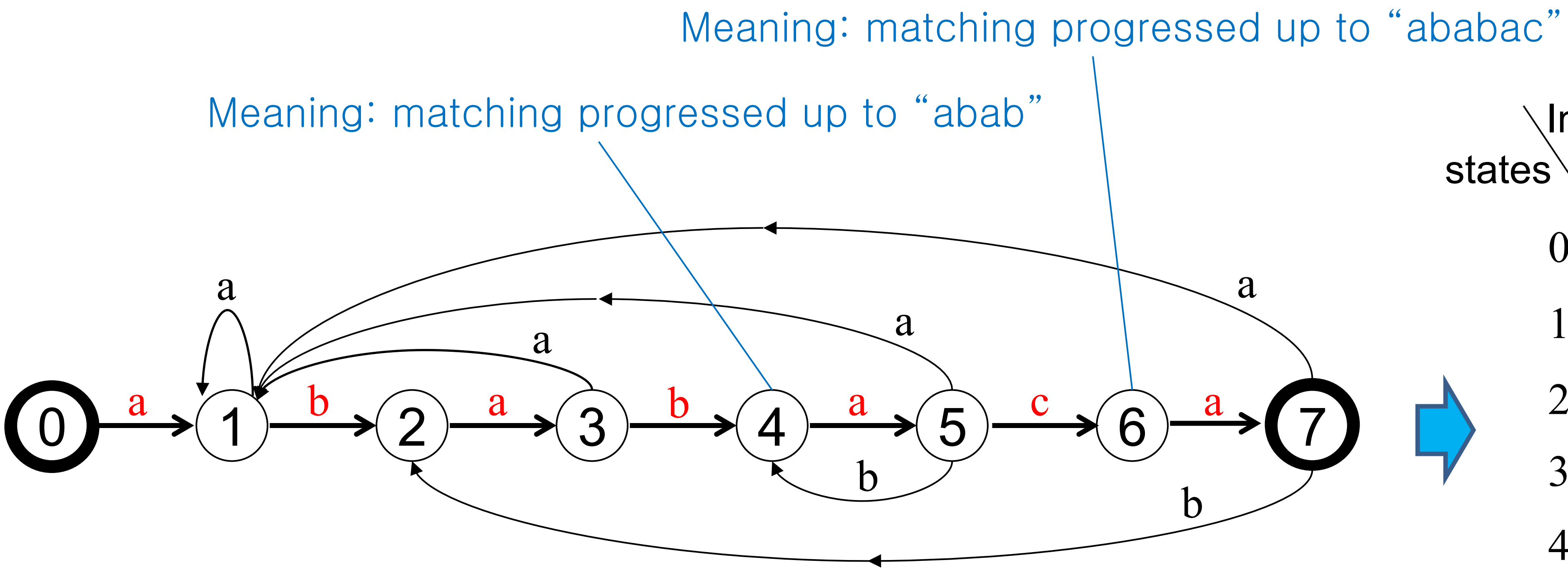


그림 12-2 원시적인 알고리즘이 비효율적인 예

오토마타를 이용한 매칭

- **오토마타**
  - 문제 해결 절차를 상태state의 전이로 나타낸 것
  - 구성 요소: (Q, q_0, A, ∑, δ)
    - Q : 상태 집합
    - q_0 : 시작 상태
    - A : 목표 상태들의 집합
    - ∑ : 입력 알파벳
    - δ : 상태 전이 함수
- **매칭이 진행된 상태들간의 관계를 오토마타로 표현한다**

ababaca를 체크하는 오토마타



S: dactababaababacabababacaagbk…

states \ Input alphabets	a	b	c	All others
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

Transition Function Table

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

알고리즘 12-2 오토마타를 이용한 매칭

```
FA_Matcher(A[], δ[][], F):
▷ F: 목표 상태 집합
▷ n: 배열 A[]의 길이
    q ← 0
    for i ← 1 to n
        q ← δ(q, A[i])
        if (q ∈ F) A[i−m+1] 자리에서 매칭이 발생했음을 알린다.
```

✓수행 시간: $\Theta(n + |\Sigma| m)$

라빈-카프 알고리즘

- **문자열 패턴을 수치로 바꾸어 문자열의 비교를 수치 비교로 대신한다**
- **수치화**
  - 가능한 문자 집합 ∑의 크기에 따라 진수가 결정된다 ← $|\Sigma| = k$

    예: $\Sigma = \{a, b, c, d, e\}$

    - $k = |\Sigma| = 5$
    - a, b, c, d, e를 각각 0, 1, 2, 3, 4에 대응시킨다
    - "cad" → $2 \cdot 5^2 + 0 \cdot 5^1 + 3 \cdot 5^0 = 28$

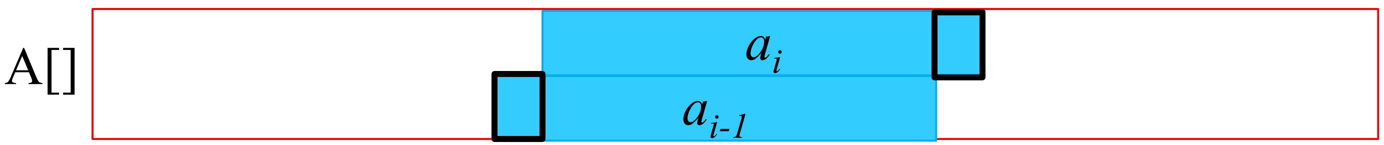
수치화 작업의 부담

A[ ]: abbafcdabafbeabebacabababacaagb…

A[0] A[i…i+m-1]

- **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 다음과 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

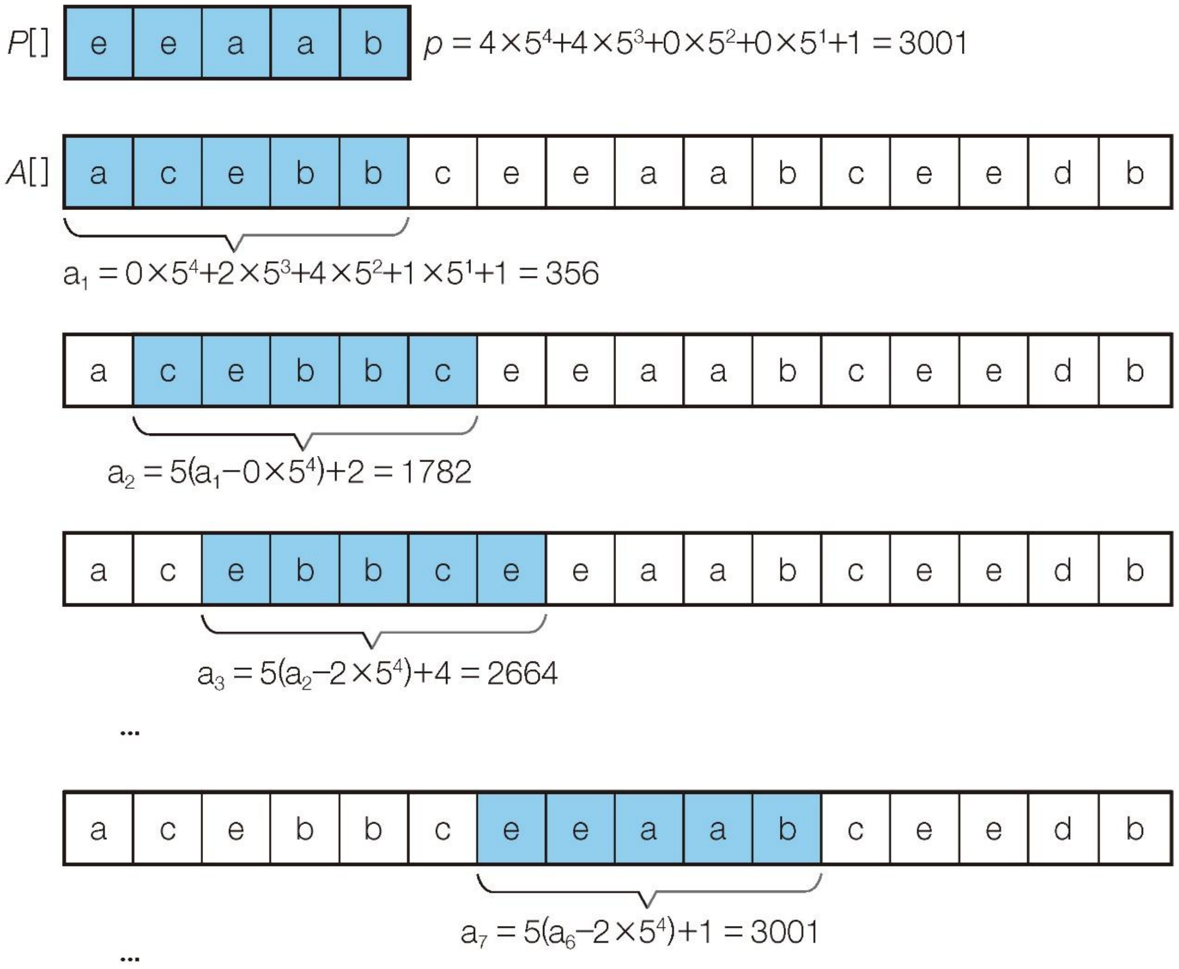


그림 12-6 수치화를 이용한 매칭 알고리즘의 작동 예

수치화를 이용해 매칭을 체크하는 알고리즘

알고리즘 12-3 수치화를 이용한 매칭

basicRabinKarp(A[], P[], d):
▷ n: 배열 A[]의 길이, m: 배열 P[]의 길이
  $p \leftarrow 0$; $a_1 \leftarrow 0$
  for $i \leftarrow 1$ to m ▷ 패턴 P[]의 수치 값과 a_1 계산
    $p \leftarrow dp + P[i]$
    $a_1 \leftarrow da_1 + A[i]$
  ❶ for $i \leftarrow 1$ to $n-m+1$
    if $(i \neq 1)$ $a_i \leftarrow d(a_{i-1} - d^{m-1}A[i-1]) + A[i+m-1]$
    if $(p = a_i)$ $A[i]$ 자리에서 매칭이 발견되었음을 알린다.

✓ 수행 시간: $\Theta(n)$

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

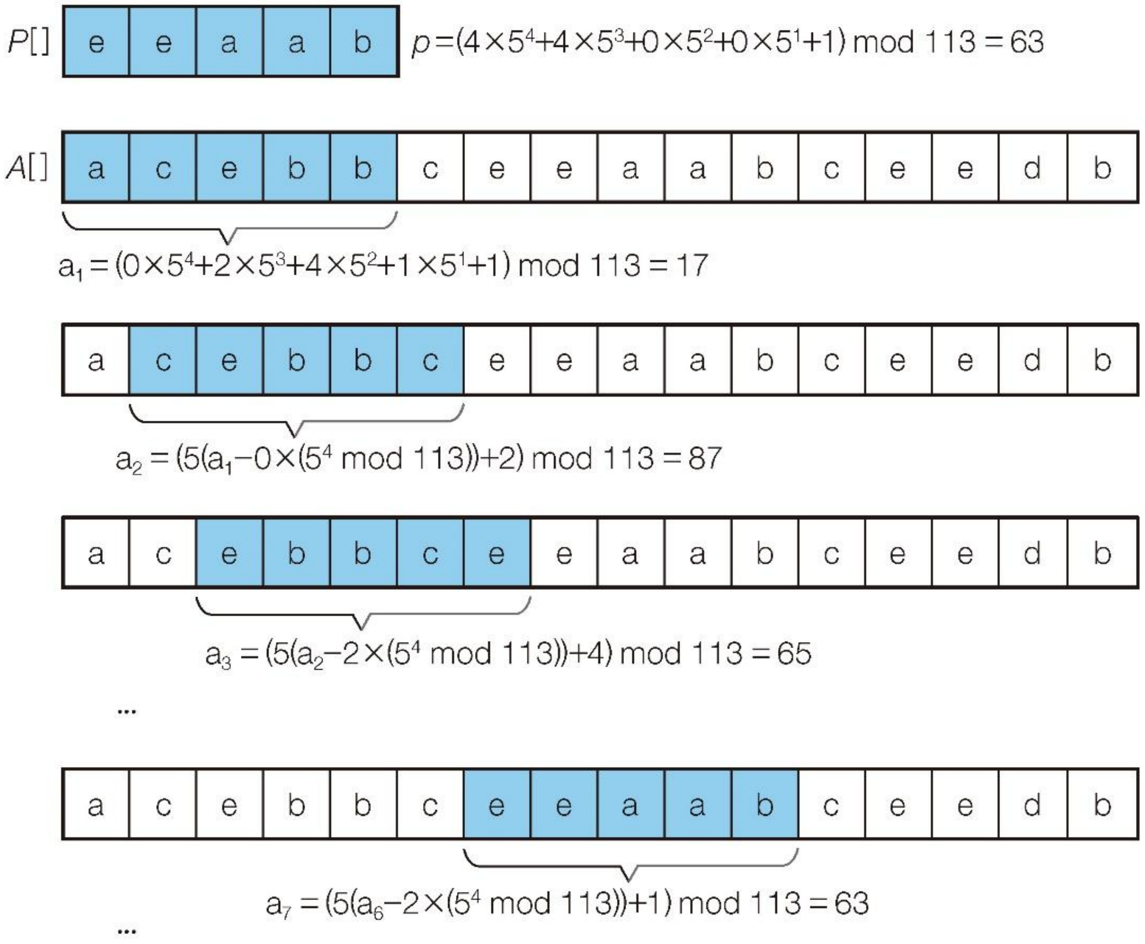


그림 12-7 **라빈-카프 알고리즘의 작동 예**

라빈-카프 알고리즘

알고리즘 12-4 라빈-카프 알고리즘

RabinKarp$(A[\,], P[\,], d, q)$:
▷ n: 배열 $A[\,]$의 길이, m: 배열 $P[\,]$의 길이
    $p \leftarrow 0$; $b_1 \leftarrow 0$
    for $i \leftarrow 1$ to m
        $p \leftarrow (dp+P[i]) \bmod q$
        $b_1 \leftarrow (db_1+A[i]) \bmod q$

    $h \leftarrow d^{m-1} \bmod q$
❶ for $i \leftarrow 1$ to $n-m+1$
        if $(i \neq 1)$ $b_i \leftarrow (d((b_{i-1}-hA[i-1]) \bmod q)+A[i+m-1]) \bmod q$
    ❷ if $(p = b_i)$
        ❸ if $(P[1...m] = A[i...i+m-1])$
                $A[i]$ 자리에서 매칭이 되었음을 알린다.

✓ 평균 수행 시간: $\Theta(n)$

KMP 알고리즘

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

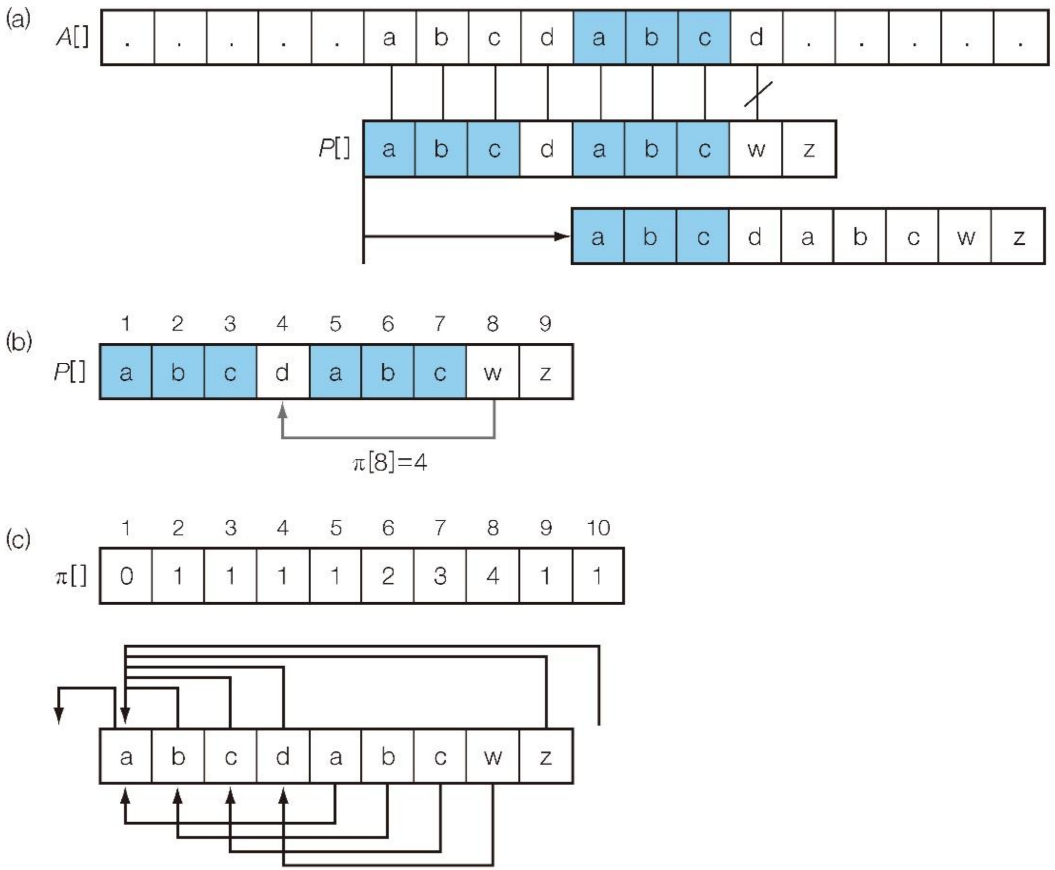


그림 12-8 KMP 알고리즘을 설명하는 예

KMP 알고리즘

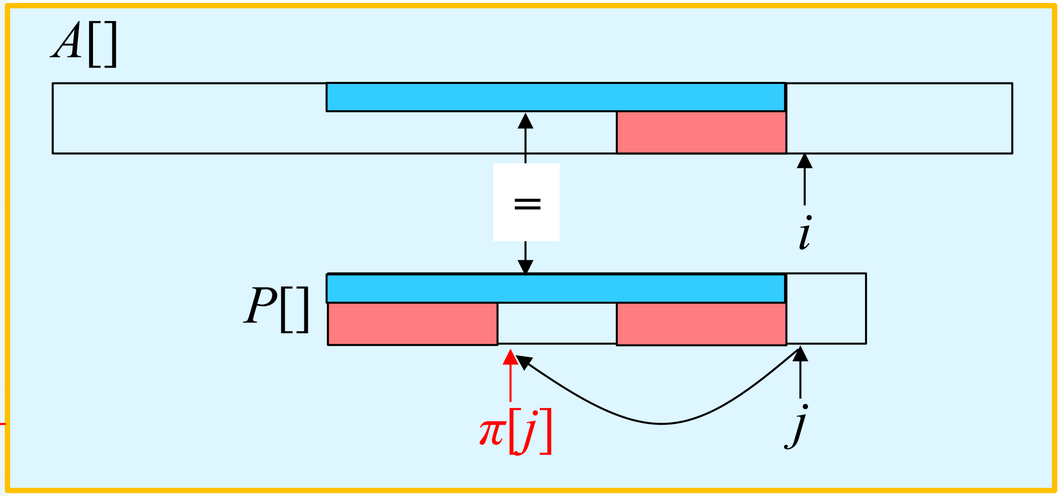
알고리즘 12-5 KMP (Knuth-Morris-Prat) 알고리즘

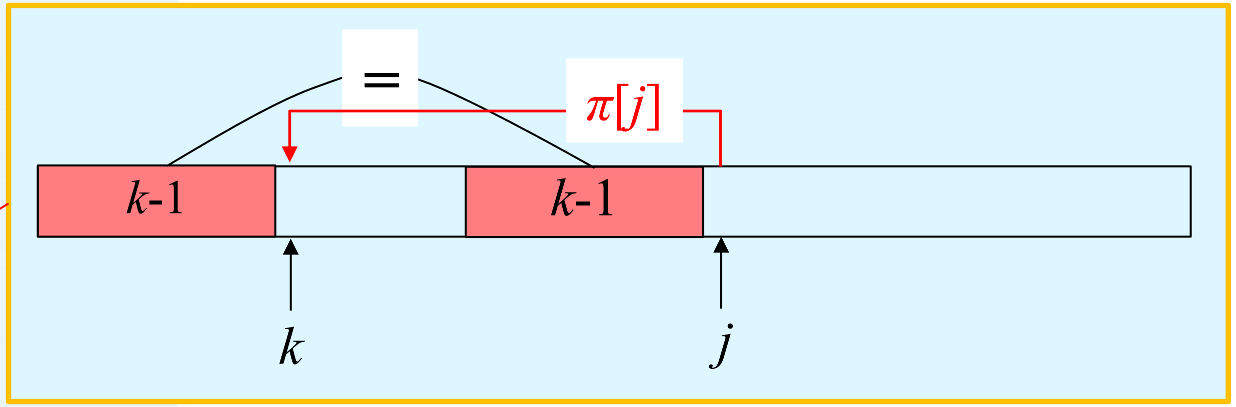
```
KMP(A[], P[]):
▷ n: 배열 A[]의 길이, m: 배열 P[]의 길이
    preprocessing(P)
    i ← 1      ▷ 본문 문자열 포인터
    j ← 1      ▷ 패턴 문자열 포인터
  ❶ while (i ≤ n)
        ❷ if (j = 0 or A[i] = P[j]) i++; j++
        ❸ else j ← π[j]
          if (j = m+1)
              A[i−m]에서 매칭되었음을 알린다.
            ❹ j ← π[j]

preprocessing(P[]):
▷ m: 배열 P[]의 길이
    j ← 1
    k ← 0
    π[1] ← 0
    while (j ≤ m)
        if (k = 0 or P[j] = P[k]) j++; k++; π[j] ← k
        else k ← π[k]
```

✓ 수행 시간: $\Theta(n)$

✓ 수행 시간: $\Theta(m)$

✓수행 시간: $\Theta(n)$





보이어-무어 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

보이어-무어 알고리즘의 작동

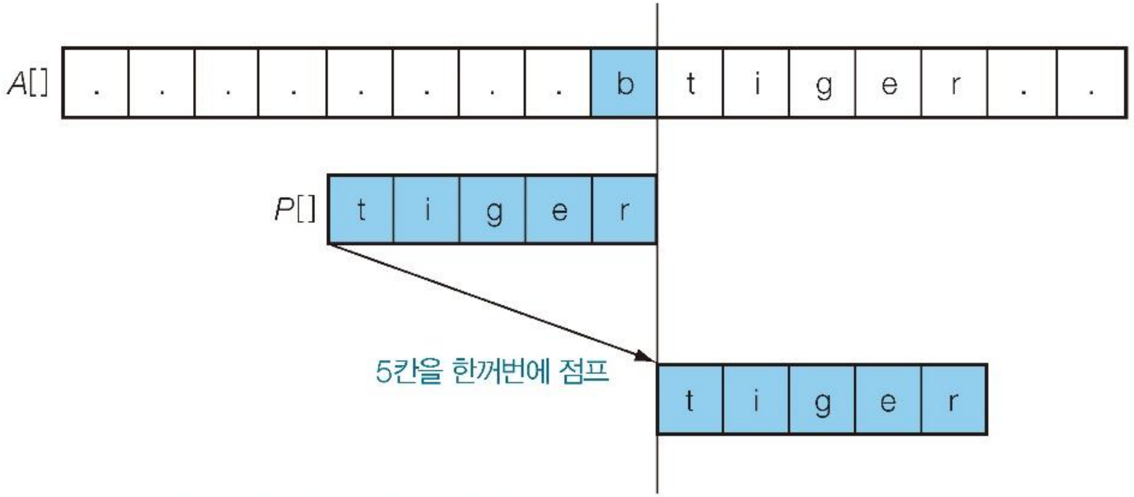


그림 12-9 패턴에 없는 문자('b')를 잘 활용하는 예

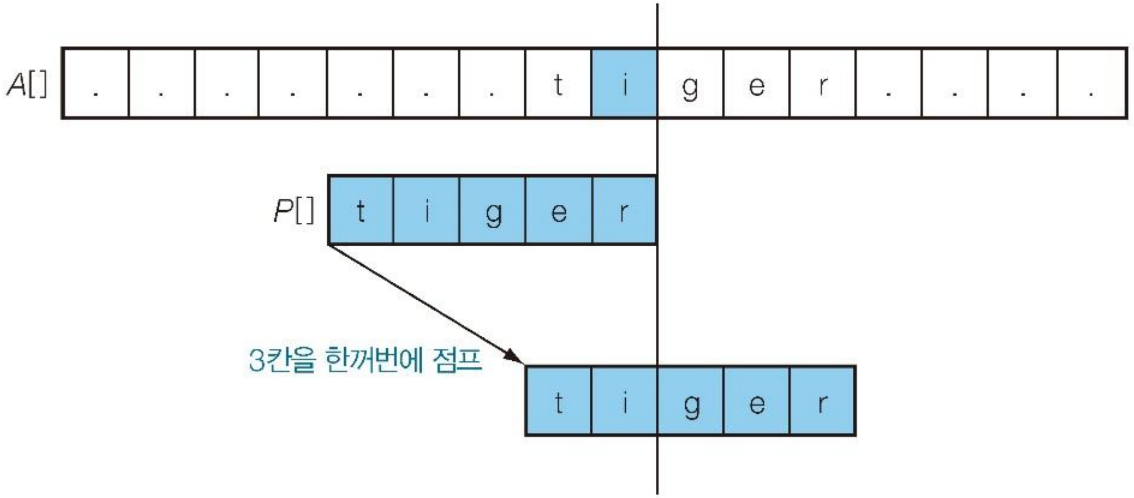


그림 12-10 패턴에 있는 문자('i')를 잘 활용하는 예

점프 정보 준비

"tiger"에 대한 점프 정보

오른쪽 끝 문자	t	i	g	e	r	기타
jump	4	3	2	1	5	5

"rational"에 대한 점프 정보

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

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

그림 12-11 **점프 정보를 만드는 예**

보이어-무어-호스풀 알고리즘

알고리즘 12-6 보이어-무어-호스풀 알고리즘

BoyerMooreHorspool(A[], P[]):
▷ n: 배열 A[]의 길이, m: 배열 P[]의 길이
▷ $jump$['a']는 기호 'a'에 대한 점프를 의미함
  ❶ computeJump(P, $jump$)
    $i \leftarrow 1$
  ❷ while ($i \le n-m+1$)
      $j \leftarrow m$; $k \leftarrow i+m-1$
    ❸ while ($j > 0$ and $P[j] = A[k]$)
        j--; k--
      if ($j = 0$) $A[i]$ 자리에서 매칭이 발견되었음을 알린다.
    ❹ $i \leftarrow i + jump[A[i+m-1]]$

불일치문자 휴리스틱과 일치접미부 휴리스틱: Optional

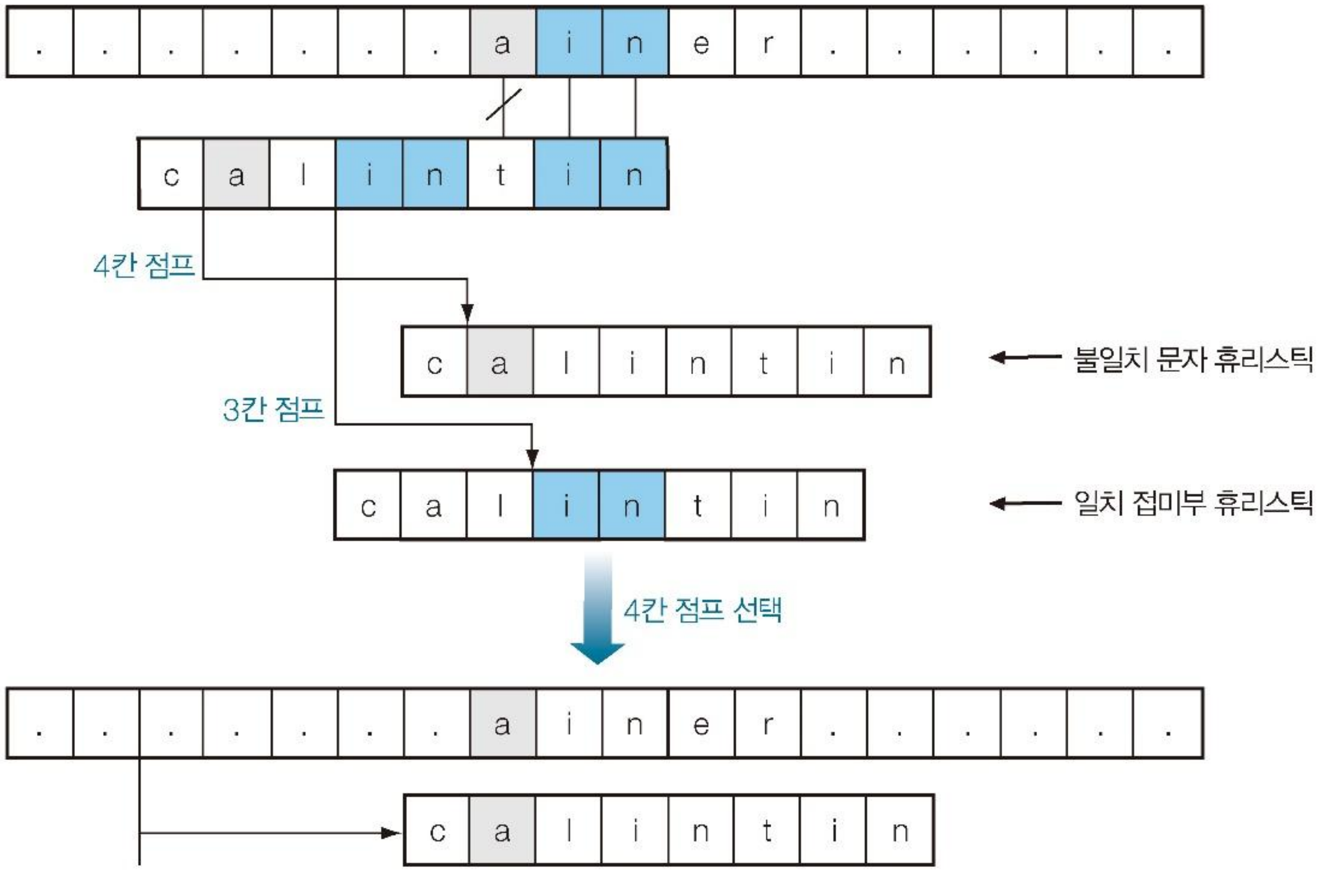


그림 12-12 일치 접미부 휴리스틱과 불일치 문자 휴리스틱 중 좋은 것 선택하기: 예 1

불일치문자 휴리스틱과 일치접미부 휴리스틱: Optional

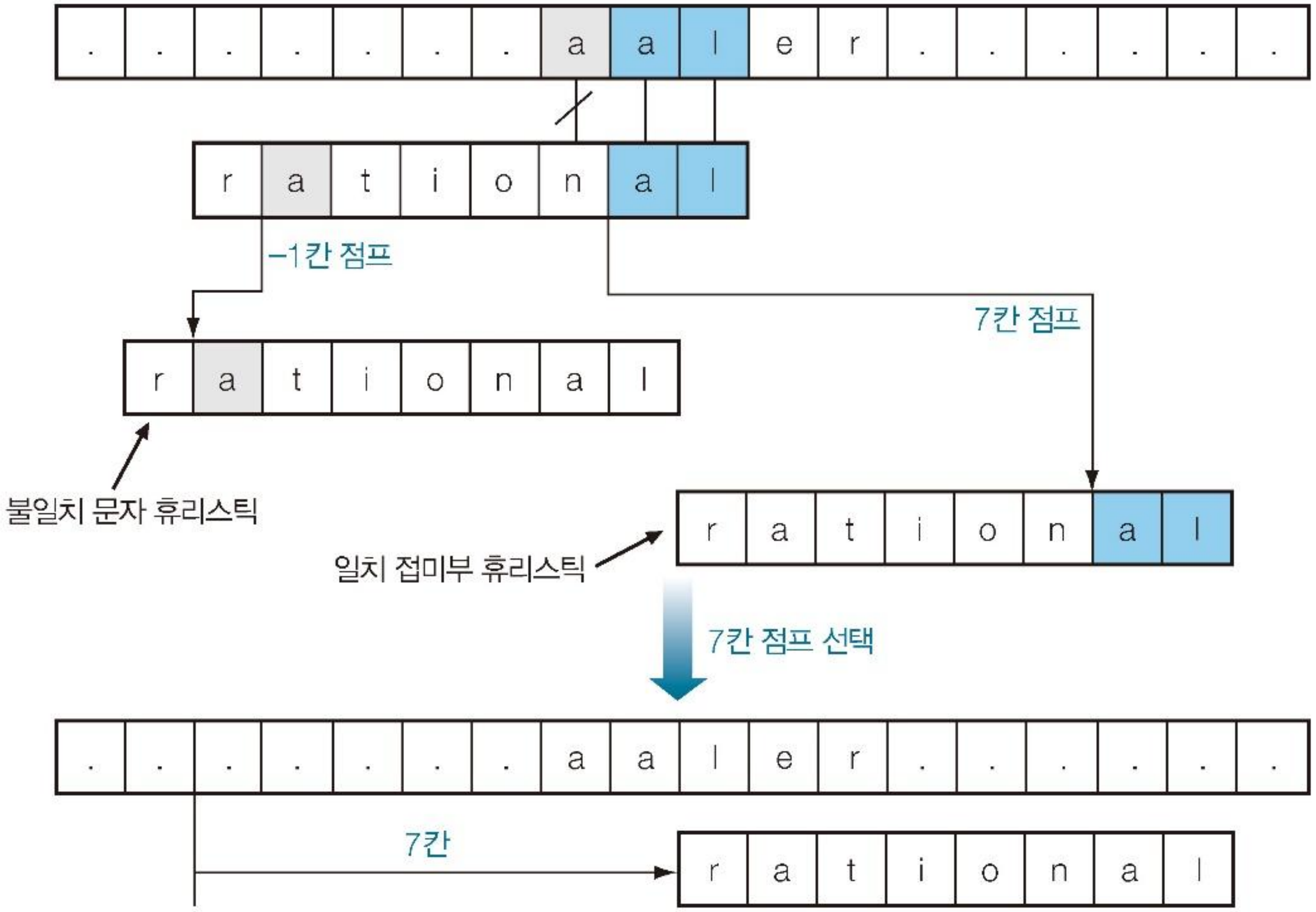


그림 12-13 일치 접미부 휴리스틱과 불일치 문자 휴리스틱 중 좋은 것 선택하기: 예 2