# 알고리즘

## 12장 문자열 매칭(string matching)

# 학습내용

1. 원시적인 매칭 방법
2. 오토마타를 이용한 매칭
3. 라빈-카프 알고리즘
4. KMP 알고리즘
5. 보이어-무어 알고리즘

# 학습목표

- 원시적인 문자열 매칭 방법의 비효율성을 이해한다.
- 오토마타를 이용한 문자열 매칭 방법을 이해한다.
- 라빈-카프 알고리즘의 수치화 과정을 이해한다.
- KMP 알고리즘을 이해하고, 오토마타를 이용한 방법과 비교시 장점을 이해한다.
- 보이어-무어 알고리즘의 개요를 이해하고, 다른 문자열 매칭 알고리즘들에 비해 어떤 장점이 있는지 이해한다.

# 문자열 매칭(string matching)

- 입력
  - A[1…*n*]: 텍스트 문자열
  - P[1…*m*]: 패턴 문자열
  - 단, *n* >> *m*      // *n*이 *m*에 비해 훨씬 크다고 가정
- 수행 작업
  - 텍스트 문자열 A[1…*n*]이 패턴 문자열 P[1…*m*]을 포함하는지를 알아본다. 또한 포함한다면 어떤 위치인지를 알아낸다.
- 예) A = “aababacccc”,  P = “aba”
  - 매칭이 일어난 위치(A의 인덱스) : 2와 4 (매칭이 두 군데서 일어나며, 매칭 시작 인덱스를 출력)

# 문자열 매칭 알고리즘

- **원시적인(naive) 매칭**
- 오토마타를 이용한 매칭
- Rabin-Karp 알고리즘
- KMP 알고리즘
- Boyer-Moore-Horspool 알고리즘

# 원시적인(naive) 매칭

naiveMatching(A[ ], P[ ])

{

▷ $n$: 배열 A의 길이, $m$: 배열 P의 길이

**for** $i \leftarrow 1$ **to** $n$-$m$+1{

**if** (P[1…$m$] = A[$i$…$i$+$m$-1]) **then**

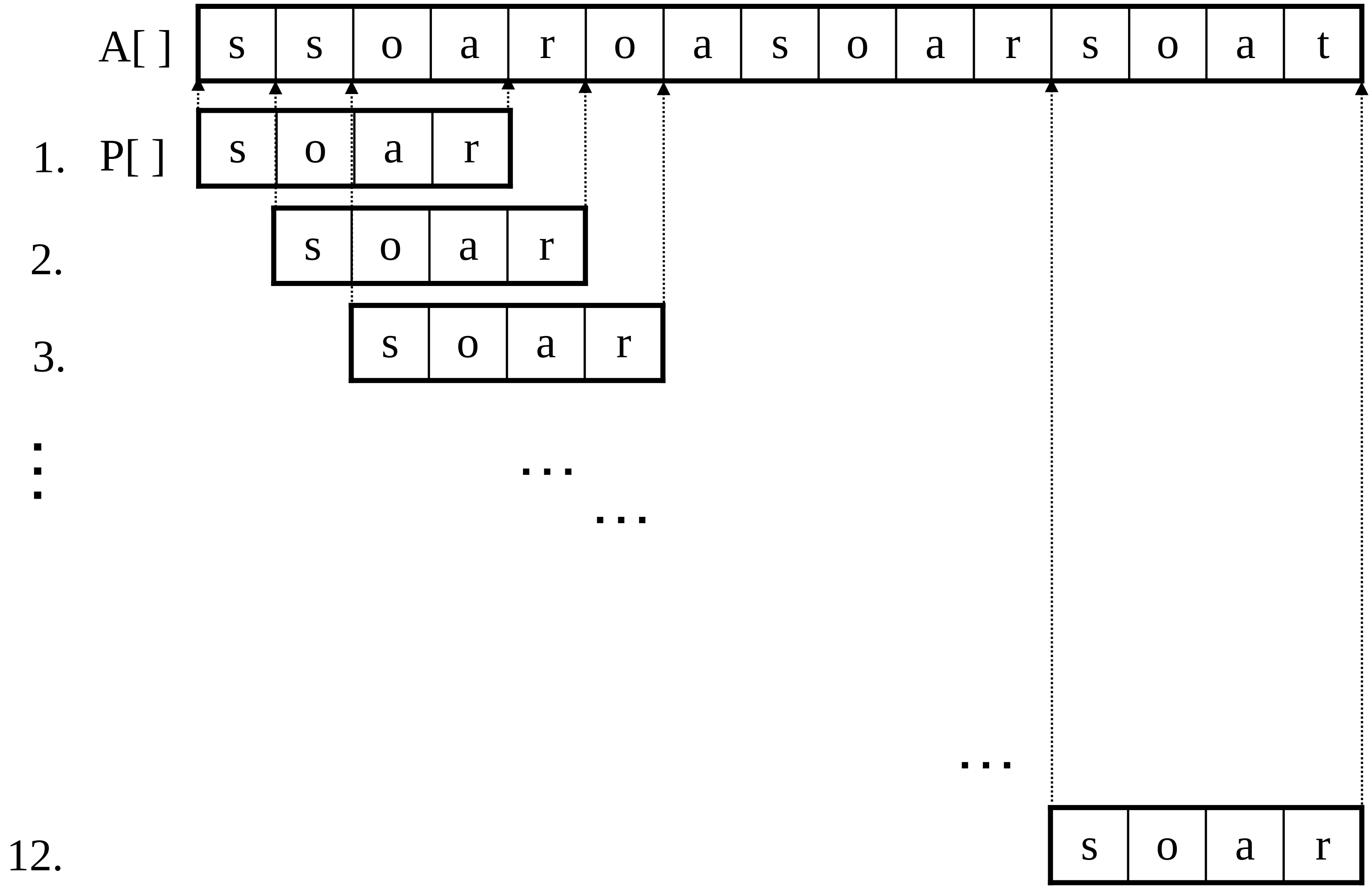
A[$i$] 자리에서 매칭이 일어났음을 알린다;

}

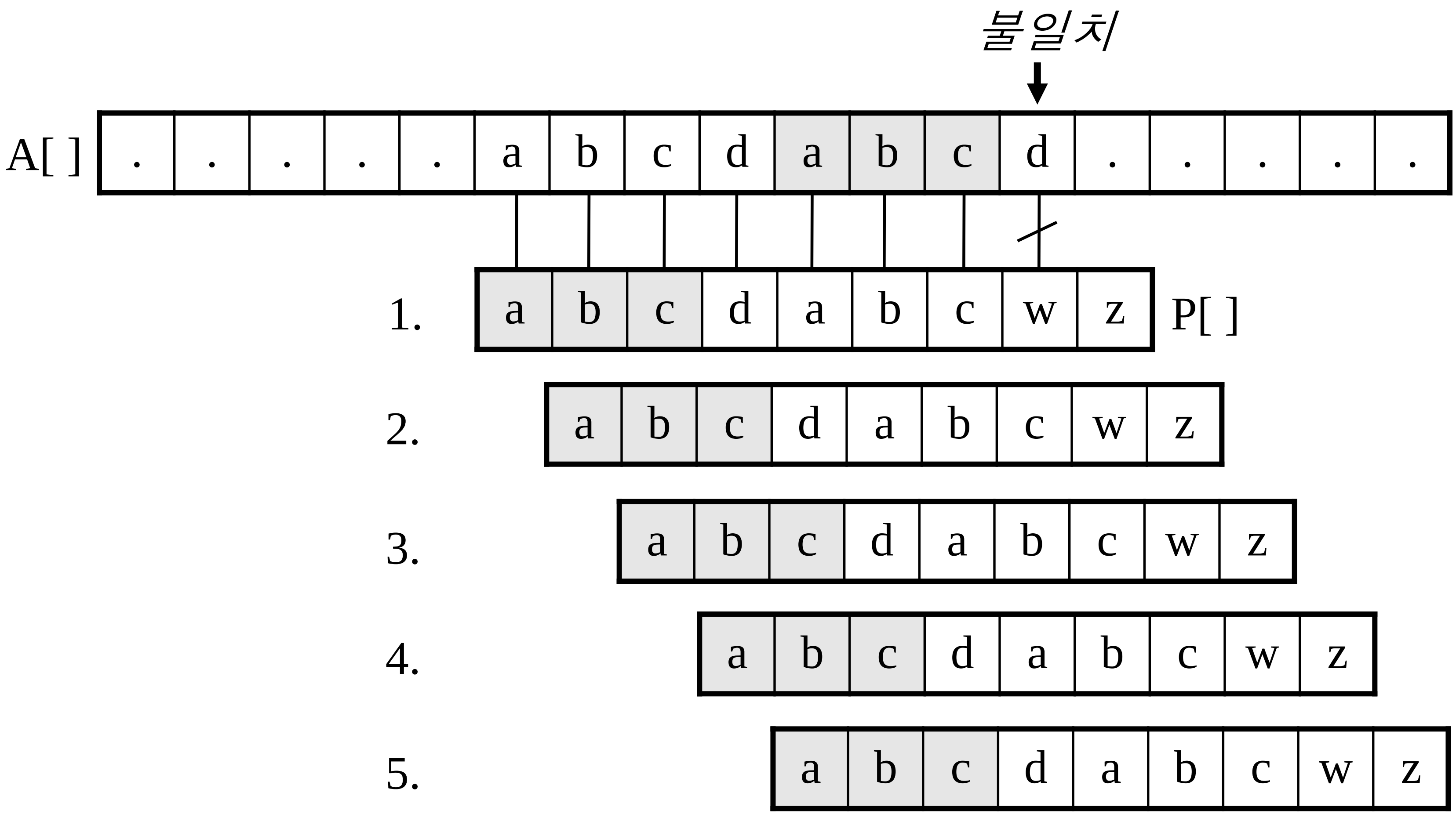
}

✓ 수행시간: $O(mn)$

# 원시적인 매칭의 작동 원리



# 원시적인 매칭이 비효율적 이유




- 앞선 매칭 과정에서 얻은 정보를 전혀 이용하지 않으므로 비효율적 : 1에서 불일치로 중단하지만, P의 앞부분 abc가 A의 두번째 abc와 일치한다는 사실을 활용할 수 있다면 2, 3, 4의 비교는 불필요하다.

# 문자열 매칭 알고리즘

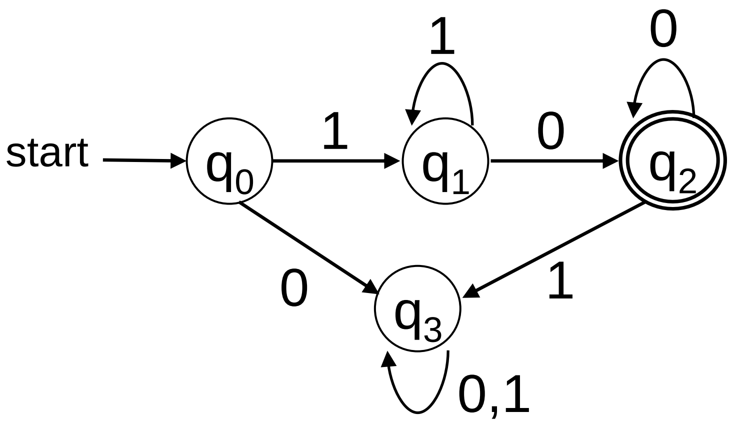
- 원시적인(naive) 매칭
- **오토마타를 이용한 매칭**
- Rabin-Karp 알고리즘
- KMP 알고리즘
- Boyer-Moore-Horspool 알고리즘

# 오토마타를 이용한 매칭

- 유한 오토마타(finite automata)
  - 문제 해결 절차를 상태 전이(state transition)로 나타냄
  - 구성 요소: $(Q, q_0, F, \sum, \delta)$
    - Q : 상태 집합
    - $q_0$ : 시작 상태
    - F : 목표 상태(최종 상태)들의 집합
    - $\sum$ : 입력 알파벳
    - $\delta$ : 상태 전이 함수 $Q \times \sum \rightarrow Q$

# 유한 오토마타

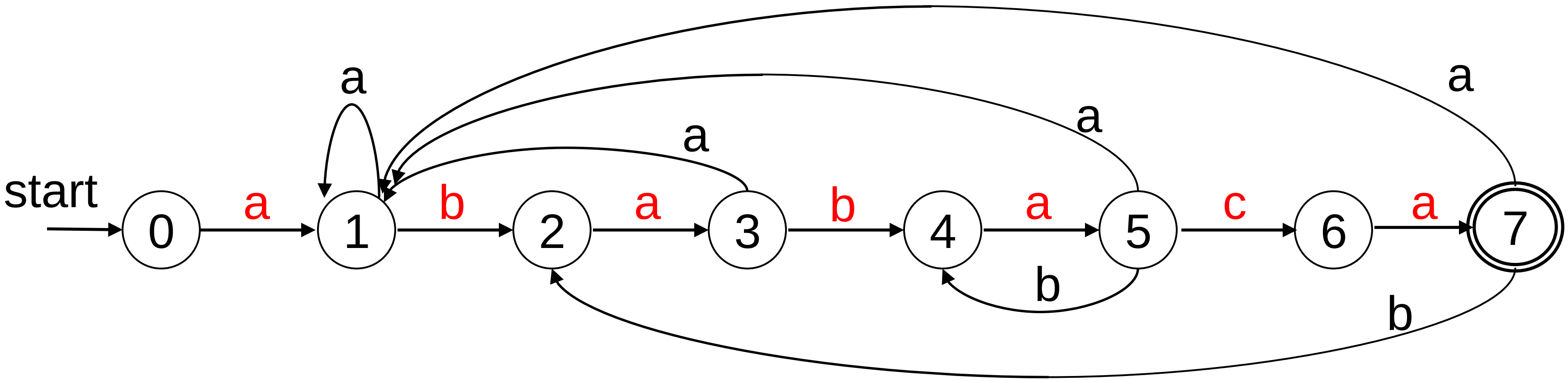
- 예)



# 오토마타를 이용한 매칭 단계

1. 전처리 작업(preprocessing)
   - 찾고자 하는 패턴에 대해 오토마타를 정의해야 함
   - P[1…m] ➔ 하나의 오토마타
     - 매칭이 어디까지 진행되었는가를 상태로 표현하고
     - 어떤 경우 어떤 상태로 전이하는가를 정의한다.
2. 매칭(matching)
   - 시작 상태에서 시작하여 텍스트 문자열의 문자를 하나씩 읽어 그 문자에 맞게 상태 전이
     - 최종 상태에 이르면 매칭된 것임
   - 이어서 텍스트 문자열을 계속해서 읽고 상태 전이도 계속함

# 1. 전처리 작업 단계
# 패턴 ababaca에 대한 오토마타




A: anbbactababaababacabababacaagbk…
P: ababaca

오토마타 $(Q, q_0, F, \sum, \delta)$
- $Q = \{0, 1, 2, 3, 4, 5, 6, 7\}$
- $q_0 = 0$
- $F = \{7\}$
- $\sum = \{a, b, c, d, \ldots, z\}$
- $\delta$ : 다음 슬라이드

오토마타에 입력 알파벳이 표시되지 않은 경우 매칭 실패로서, 시작 상태로 전이한다고 본다.

# 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 |

# 2. 매칭 단계 – 매칭 알고리즘

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

- 위의 알고리즘 수행 시간 = $\Theta(n)$
- 상태전이함수 구성 시간 = 가장 효율적이라고 볼 때 $\Theta(|\Sigma|m)$

✓오토마타를 이용한 매칭 총 수행시간: $O(n + |\Sigma|m)$

# 문자열 매칭 알고리즘

- 원시적인(naive) 매칭
- 오토마타를 이용한 매칭
- **Rabin-Karp 알고리즘**
- KMP 알고리즘
- Boyer-Moore-Horspool 알고리즘

# Rabin-Karp 알고리즘

- 문자열 패턴을 수치로 바꾸어 문자열의 비교를 수치 비교로 대신한다.
- 수치화
  - 가능한 문자 집합 ∑의 크기에 따라 진수가 결정된다.
  - 예: ∑ = {a, b, c, d, e, f, g, h, i, j}
    - |∑| = 10
    - a, b, c, … , j 를 각각 0, 1, 2, … , 9 에 대응시킨다.
    - 문자열 “cad”를 수치화하면 $2*10^2+0*10^1+3*10^0 = 203$
- |∑|를 d 라고 하자.

# 문자열 수치화 예

- ∑ = {a, b, c, d, e}
  - |∑| = 5
  - a, b, c, d, e 를 각각 0, 1, 2, 3, 4 에 대응시킨다.
  - 문자열 “cad”를 수치화하면 $2*5^2+0*5^1+3*5^0 = 53$
  - 문자열 “bec”를 수치화 하면 $1*5^2+4*5^1+2*5^0 = 47$
  - 문자열 “eca”를 수치화 하면 $4*5^2+2*5^1+0*5^0 = 110$
- 문자열 수치화를 이용한 문자열 매칭 예

A = “becad”, P = “cad”

# 해결해야 하는 문제점

- 문자열 수치화를 이용하여 문자열 매칭을 수행하려면 다음과 같은 문제점을 해결해야 한다.
  - 수치화 작업의 부담
  - 수치가 커져 오버플로우 발생 가능성

➔Rabin-Karp 알고리즘은 이 두가지 문제점을 해결하여 문자열 매칭을 수행한다.

# 수치화 작업의 부담

- P[1...*m*]에 대응하는 수치 $p$의 계산
  - $p$ = ((...((P[1]*d)+P[2])*d + …)*d+P[$m$ -1])*d+P[$m$]

  예) “edabc” ➔ $p = 4*5^4 + 3*5^3 + 0*5^2 + 1*5^1 + 2$

  $= ((((4*5)+3)*5+0)*5+1)*5+2$
- A[$i$…$i$+$m$-1]에 대응하는 수치 $a_i$ 의 계산
  - $a_i$ = ((...((A[$i$]*d)+A[$i$+1])*d + …)*d+A[$i$+$m$-2])*d+A[$i$+$m$-1]

  예) “<u>dceda</u>bcc” ➔ $a_1 = 3*5^4 + 2*5^3 + 4*5^2 + 3*5^1 + 0$

  $= ((((3*5)+2)*5+4)*5+3)*5+0$

  ➔ $p$와 $a_i$를 비교하여 값이 같으면 i에서 매칭
- 문제점
  - $a_i$ 계산에 $\Theta(m)$의 시간이 든다.
  - $a_i$를 일일이 계산하는 경우 A[1…$n$] 전체에 대한 비교는 $\Theta(mn)$이 소요된다. ➔ 원시적인 매칭에 비해 나은 게 없다.

# 수치화 작업의 부담

- 다행히, $m$의 크기에 상관없이 아래와 같이 계산할 수 있으므로 이 문제는 해결된다.
  - $a_i = d(a_{i-1} - d^{m-1}A[i\text{-}1]) + A[i+m\text{-}1]$
  - $d^{m-1}$은 반복 사용되므로 미리 한번만 계산해 두면 된다.
  - 곱셈 2회, 덧셈 2회로 충분

예) "**d**ceda**b**cc"

$a_1 = 3*5^4 + 2*5^3 + 4*5^2 + 3*5^1 + 0$ ▷ "**d**ceda"

$a_2 = 5*(a_1 - \mathbf{5^4*A[1]}) + \mathbf{A[6]}$

$= 5*(a_1 - \mathbf{5^4*3}) + \mathbf{1}$

즉, dceda를 나타내는 수치로부터 cedab를 나타내는 수치를 계산하는 데 상수 시간이 걸림

# 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 |

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

…

이것은 아직 라빈-카프 알고리즘이 아님

# 수치화를 이용해 매칭을 체크하는 알고리즘

basicRabinKarp(A, P, d) ▷ $n$ : 배열 A의 길이, $m$ : 배열 P의 길이
{
  p ← 0; $a_1$ ← 0;
  **for** i ← 1 **to** $m$ {      ▷ 패턴 P의 수치값 p와 $a_1$ 계산
    p ← dp + P[i];
    $a_1$ ← d$a_1$ + A[i];
  }
  **if** (p = $a_1$) **then** A[1] 자리에서 매칭이 일어났음을 알린다;
  **for** i ← 2 **to** $n$-$m$+1{
    $a_i \leftarrow d(a_{i-1} - d^{m-1}A[i-1]) + A[i+m-1]$;
    **if** (p = $a_i$) **then** A[i] 자리에서 매칭이 일어났음을 알린다;
  }
}

✓ 총 수행시간: $\Theta(n)$

# basicRabinKarp 알고리즘의 문제점

- 문자 집합 크기 d와 패턴의 길이 $m$에 따라 $a_i$가 매우 커질 수 있다.
  - 심하면 컴퓨터 레지스터의 용량 초과
  - 오버플로우 발생
- 해결책
  - 나머지 연산(modulo)을 사용하여 $a_i$의 크기를 제한한다.
  - $\boxed{a_i = d(a_{i-1} - d^{m-1}A[i-1]) + A[i+m-1]}$ 대신

    $\boxed{b_i = (d(b_{i-1} - (d^{m-1} \textbf{ mod } \textbf{q})A[i-1]) + A[i+m-1]) \textbf{ mod } \textbf{q}}$ 사용
  - q를 충분히 큰 소수(prime number)로 정하되, dq가 레지스터에 수용될 수 있는 크기로 한다.

# 나머지 연산을 이용한 매칭 예

$d^x$도 너무 커지므로 실제로 구현할 때는 이 값에도 mod 연산을 함

P[ ]

| | | | | |
|---|---|---|---|---|
| e | e | a | a | b |

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

A[ ]

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

$b_1 = (0*5^4+2*5^3+4*5^2+1*5^1+1) \bmod 113 = 17$

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

$b_2 = (5(b_1-0*(5^4 \bmod 113))+2) \bmod 113 = 87$

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

$b_3 = (5(b_2-2*(5^4 \bmod 113))+4) \bmod 113 = 65$

…

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

$b_7 = (5(b_6-2*(5^4 \bmod 113))+1) \bmod 113 = 63$

…

이것이 제대로 된 라빈-카프 알고리즘

# 라빈-카프 알고리즘

RabinKarp(A, P, d, q) ▷ $n$ : 배열 A의 길이, $m$ : 배열 P의 길이
{
  h ← 1;
  **for** i ← 1 **to** $m$-1
    h ← dh mod q; ▷ h 계산(= $d^{m-1}$ mod q)

  p ← 0; $b_1$ ← 0;
  **for** i ← 1 **to** $m$ {
    p ← (dp + P[i]) mod q; ▷ 패턴 P의 수치값 p 계산
    $b_1$ ← ($db_1$ + A[i]) mod q; ▷ $b_1$ 계산
  }

  **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** ▷ 진짜 매칭인지 알아봄
        A[i] 자리에서 매칭이 일어났음을 알린다;
  }
}

✓ 매칭 횟수가 상수 번이면 $\Theta(n)$ ➔ 평균 수행시간 $\Theta(n)$

# 라빈-카프 알고리즘

- 최악의 경우 $\Theta(mn)$
  - 모든 문자가 동일한 경우. 예를 들어

$$A = \underbrace{\text{aaaaa}\ldots\ldots\ldots\ldots\ldots\text{aaa}}_{n} = \text{a}^{n}$$

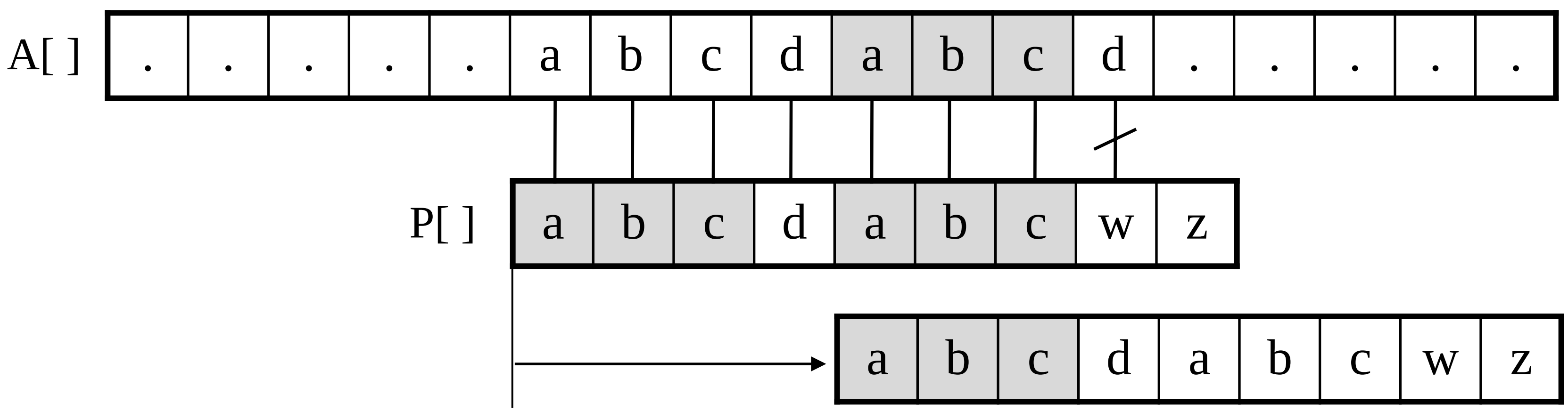
$$P = \underbrace{\text{aaaa}\ldots\ldots\ldots\text{aa}}_{m} = \text{a}^{m}$$

# 문자열 매칭 알고리즘

- 원시적인(naive) 매칭
- 오토마타를 이용한 매칭
- Rabin-Karp 알고리즘
- **KMP 알고리즘**
- Boyer-Moore-Horspool 알고리즘

# KMP(Knuth-Morris-Pratt) 알고리즘

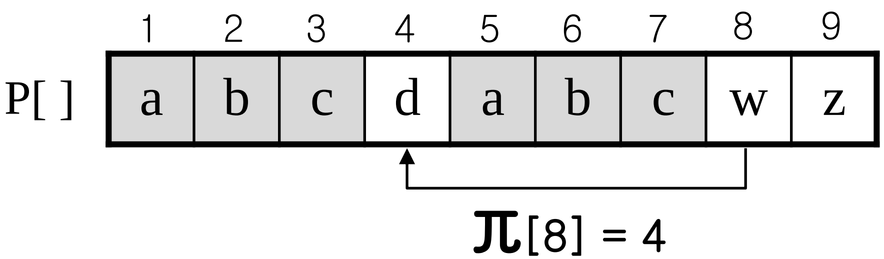
- 오토마타를 이용한 매칭과 동기가 유사
  - 매칭에 실패했을 때 돌아갈 상태를 준비해둔다.
- 오토마타를 이용한 매칭보다 전처리 작업이 단순




- P[8] 매칭에 실패했으므로 실패한 텍스트 위치에서 P[4]부터 다시 매칭 시작

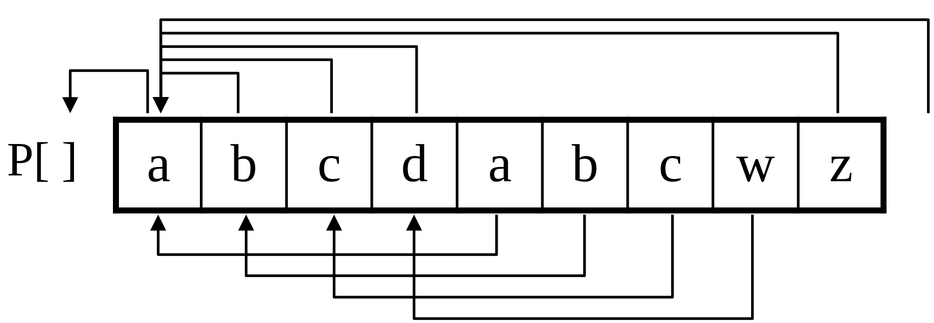
# KMP 알고리즘의 전처리(preprocessing)

## 매칭이 실패했을 때 돌아갈 곳을 준비하는 작업




- 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 곳에 대한 정보 π를
준비해 둔다.

# KMP 알고리즘

```
KMP(A[ ], P[ ]) ▷ n: 배열 A의 길이, m: 배열 P의 길이
{
    preprocessing(P);         // 수행시간 Θ(m)

    i ← 1; ▷ 본문 문자열 포인터
    j ← 1; ▷ 패턴 문자열 포인터

    while (i ≤ n) {
        if (j = 0 or A[i] = P[j])
            then { i++; j++; }
            else  j ← π[j];
        if (j = m+1) then {
            A[i-m]에서 매칭이 일어났음을 알린다;
            j ← π[j];
        }
    }
}
```

✓수행시간: $\Theta(n)$

# KMP 전처리 작업 알고리즘

```
preprocessing(P[ ])
{
    j ← 1;
    k ← 0;
    π[1] ← 0;

    while (j ≤ m) {
        if (k = 0 or P[j] = P[k])
            then { j++;  k++;  π[j] ← k; }
            else  k ← π[k];
    }
}
```

✓수행시간: $\Theta(m)$

# 문자열 매칭 알고리즘

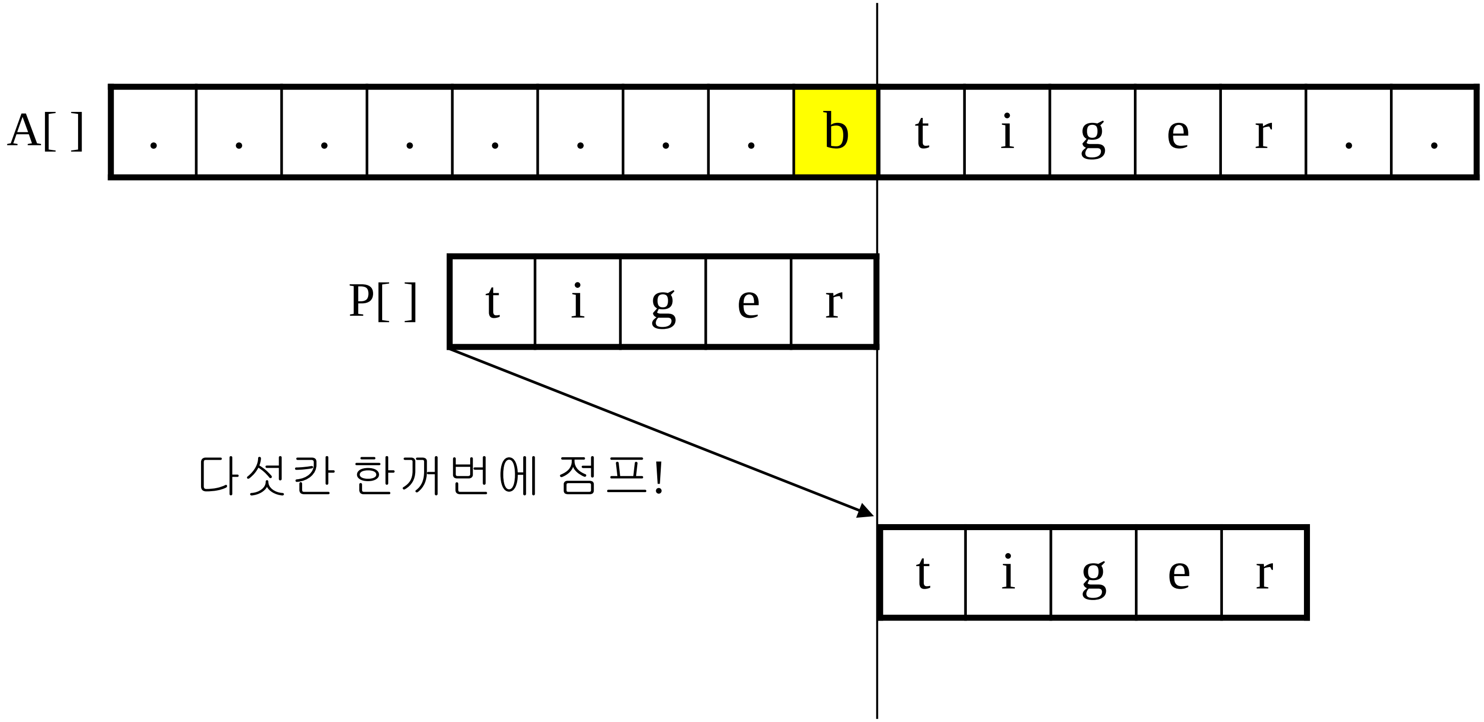
- 원시적인(naive) 매칭
- 오토마타를 이용한 매칭
- Rabin-Karp 알고리즘
- KMP 알고리즘
- **Boyer-Moore-Horspool 알고리즘**

# Boyer-Moore-Horspool 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 비교
  - 매치될 가능성이 없으면 점프
- 보이어-무어-호스풀 알고리즘
  - 보이어-무어 알고리즘에서 까다로운 부분을 약식으로 처리한 알고리즘으로서, 약식으로 처리해도 전체 성능에는 거의 영향을 주지 않음
  - 라빈-카프 알고리즘, KMP 알고리즘보다 평균적으로 빠름

# Motivation

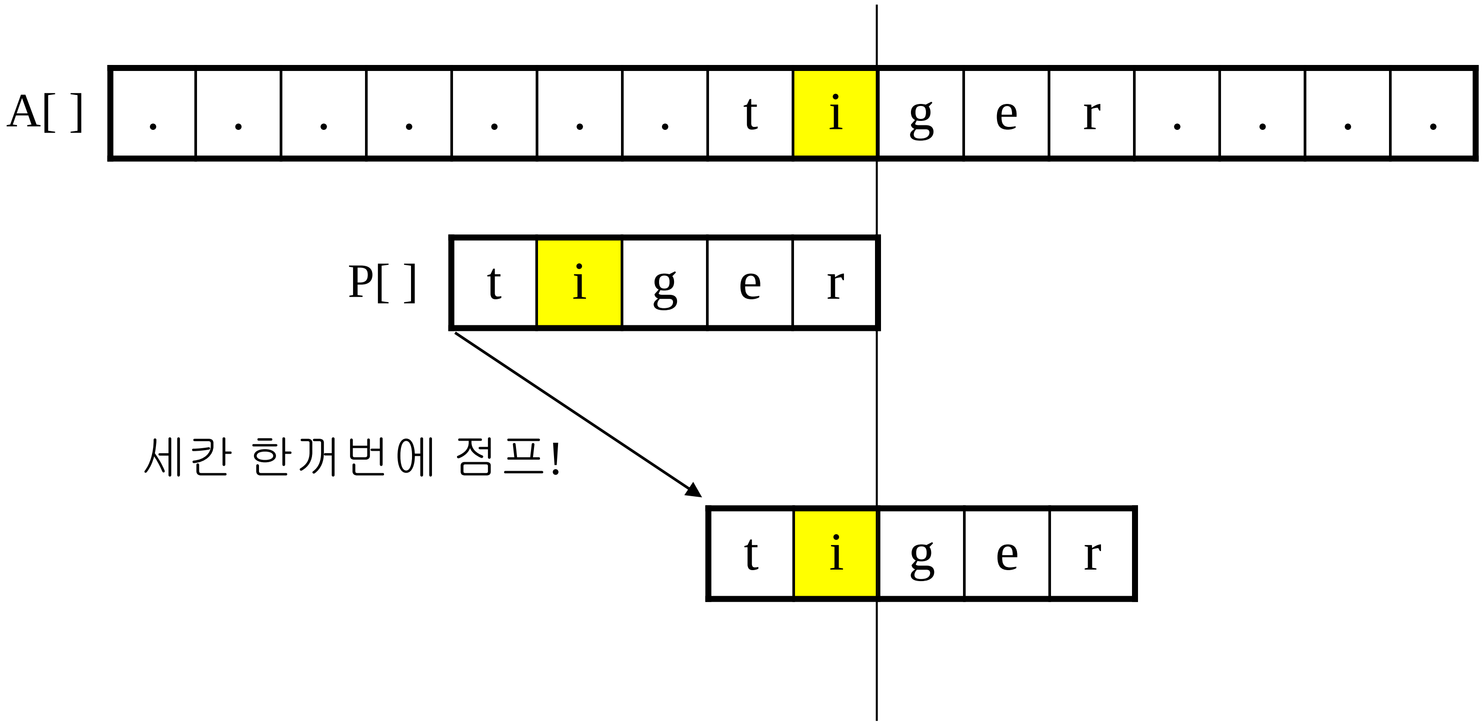
상황: 텍스트의 b와 패턴의 r을 비교하여 실패했다.




- ✓ 관찰: 패턴에 문자 b가 없으므로
  패턴 전체가 텍스트의 b를 뛰어넘어도 된다.

# Motivation

상황: 텍스트의 i와 패턴의 r을 비교하여 실패했다.




✓ 관찰: 패턴에서 i가 r의 3번째 왼쪽에 나타나므로 패턴이 한꺼번에 3칸을 뛰어 넘어도 된다.

# 점프 정보 준비(preprocessing)

- 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 |

# 보이어-무어-호스풀 알고리즘

```
BoyerMooreHorspool(A[ ], P[ ]) ▷ n : 배열 A의 길이, m : 배열 P의 길이
{
    computeJump(P, jump);          // preprocessing : 수행시간 Θ(m)
    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 일어났음을 알린다;
        i ← i + jump[A[i + m − 1]];     //  jump 정보 얻는 시간 Θ(1)
    }
}
```

✓최악의 경우 수행시간: $\Theta(mn)$

✓최선의 경우 수행시간: $\Theta(n/m)$

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

# 보이어-무어-호스풀 알고리즘

- 최악의 경우 $\Theta(mn)$

  모든 문자가 동일한 경우. 예를 들어

  A = aaaaa…...........aaa = $a^n$

  P = aaaa......aa = $a^m$

- 최선의 경우 $\Theta(n/m)$

  예를 들어

  A = abcdybbbbkcccctddddx

  P = abcde

# 예 1

패턴 “tiger”에 대한 점프 정보

| 오른쪽 끝문자 | t | i | g | e | r | 기타 |
|---|---|---|---|---|---|---|
| jump | 4 | 3 | 2 | 1 | 5 | 5 |

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | t | i | g | e | a | t | i | t | i | g | e | r | t | i | t | e | r | a |

| | | | | | |
|---|---|---|---|---|---|
| P | t | i | g | e | r |

# 예 2

패턴 "tiger"에 대한 점프 정보

| 오른쪽 끝문자 | t | i | g | e | r | 기타 |
|---|---|---|---|---|---|---|
| jump | 4 | 3 | 2 | 1 | 5 | 5 |

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | t | i | g | e | a | t | i | g | e | d | t | o | d | a | y | e | r | a |

| | | | | | |
|---|---|---|---|---|---|
| P | t | i | g | e | r |

# 요약

- 원시적인 매칭 알고리즘은 텍스트의 각 위치에서 시작해 패턴 문자열과 일치하는지 체크하는 방법으로, 수행 시간은 O($mn$)이다.
- 오토마타를 이용하는 알고리즘은 매칭 과정에서 불일치가 일어났을 때 처음부터 다시 비교하지 않고 문맥상의 정보를 이용해 중간부터 비교할 수 있도록 한다.
- 라빈-카프 알고리즘은 패턴을 수치화해(문자열 비교를 수치 비교로 전환해) 문자열 매칭을 수행한다.

# 요약

- KMP 알고리즘은 매칭 과정에서 불일치가 일어났을 때 처음부터 다시 비교하지 않고 중간부터 비교할 수 있도록 되돌아 갈 위치를 배열로 나타낸다.
- 보이어-무어-호스풀 알고리즘은 패턴의 뒷부분에 대응되는 텍스트의 문자를 이용해 텍스트를 보지 않고도 뛰어넘을 수 있게 한다. 최악의 경우 시간은 O($mn$)이지만 평균적으로 가장 좋은 성능을 보인다.

# 문자열 매칭 알고리즘

텍스트 길이 $n$
패턴 길이 $m$

| 알고리즘 | preprocessing time | matching time | 특징 |
|---|---|---|---|
| Naive(brute force) | 0 | $O(mn)$ | 매우 원시적 |
| Finite automaton | $O(m\|\Sigma\|)$ | $\Theta(n)$ | 오토마타 이용<br>$\Sigma$ : 입력 알파벳 |
| Rabin-Karp | $\Theta(m)$ | 최악 $\Theta(mn)$<br>최선 $\Theta(n)$<br>평균 $\Theta(n)$ | 패턴을 수치화 |
| Knuth-Morris-Pratt | $\Theta(m)$ | $\Theta(n)$ | 부분패턴에 깃든<br>효용성을 최대한 이용 |
| Boyer-Moore-Horspool | $\Theta(m)$ | 최악 $\Theta(mn)$<br>최선 $\Theta(n/m)$<br>평균 $\Theta(n)$보다는<br>$\Theta(n/m)$에 가까움 | 텍스트 문자열을 보지<br>않고 점프할 수 있는<br>기회를 최대화 |