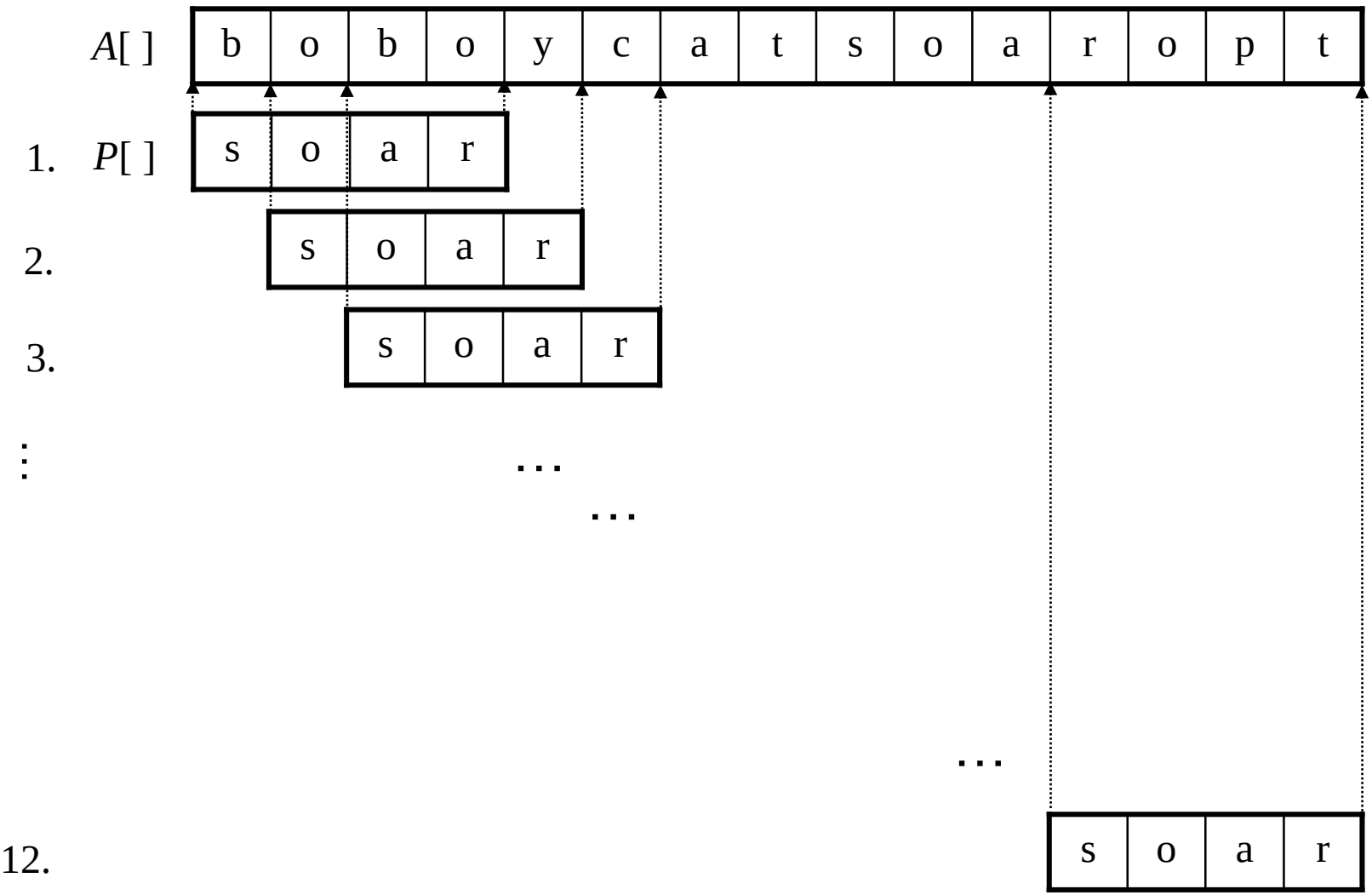
String Matching

String Matching

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

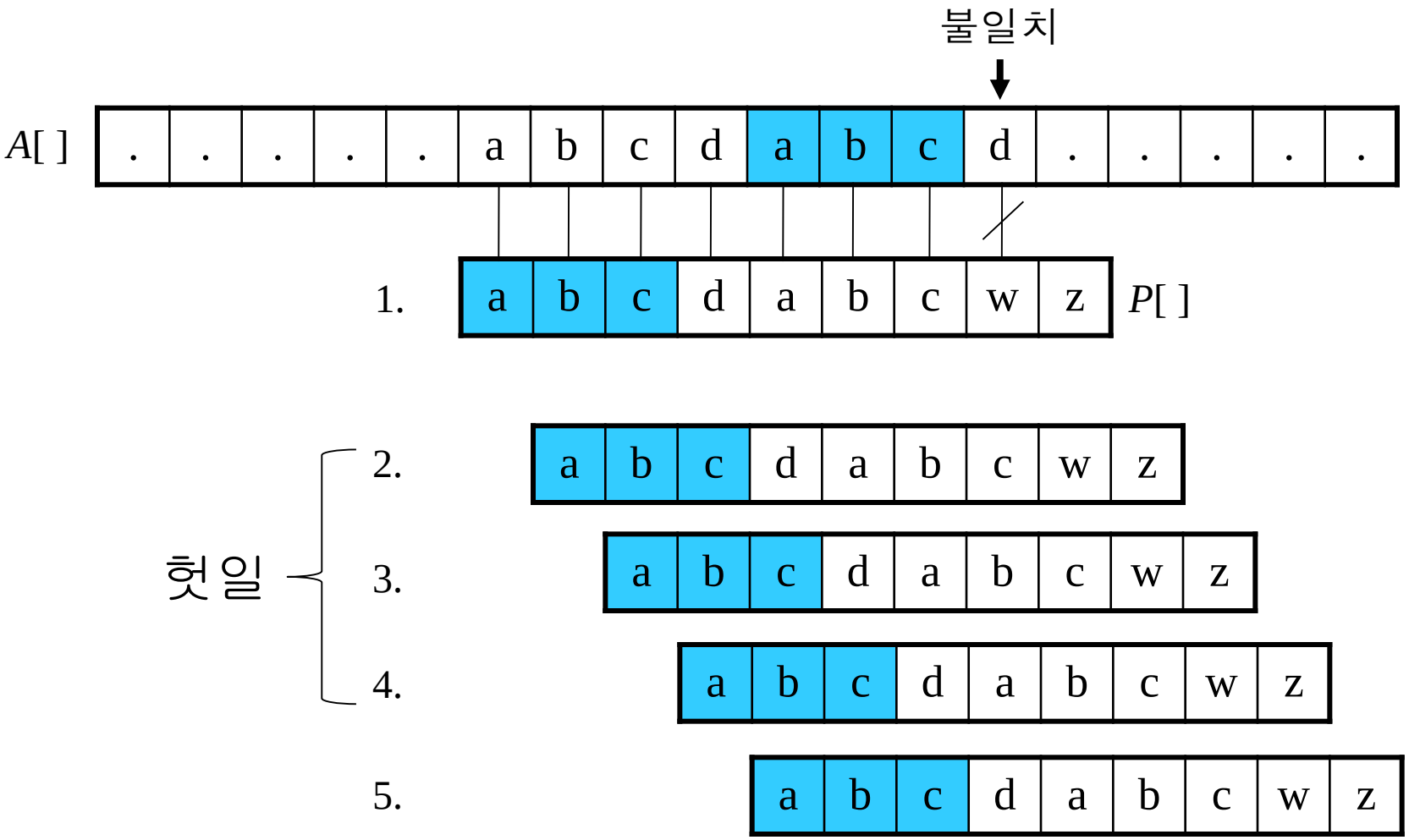
원시적인 매칭



```
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: $O(mn)$

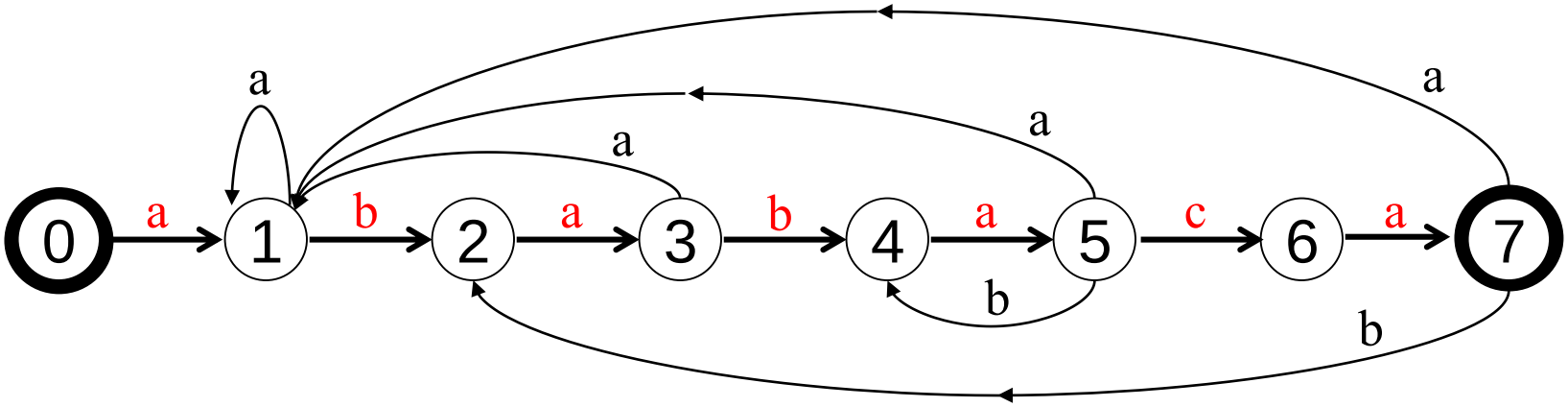
원시적인 매칭이 비효율적인 예



오토마타를 이용한 매칭

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

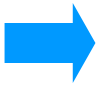
ababaca를 체크하는 오토마타



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

오토마타 만들기

```
FA-Generater (P[ ], ∑)
▷ P[1…m]: 패턴
{
    for q ← 0 to m {
        for each a ∈ ∑ {
            k ← min(m+1, q+2);
            repeat k-- ;
            until (P[1…k]가 P[1…q]·a의 suffix)   ▷ x·a = xa
            δ(q, a) ← k;
        }
}
```

✓ 수행시간: $\Theta(|\sum|m)$: 좀 영리한 아이디어 필요(뒤의 KMP와 관련)

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;
    }
}
```

- ✓ 수행시간: $\Theta(n)$
- ✓ 총 수행시간: $\Theta(n + |\Sigma|m)$

라빈-카프Rabin-Karp 알고리즘

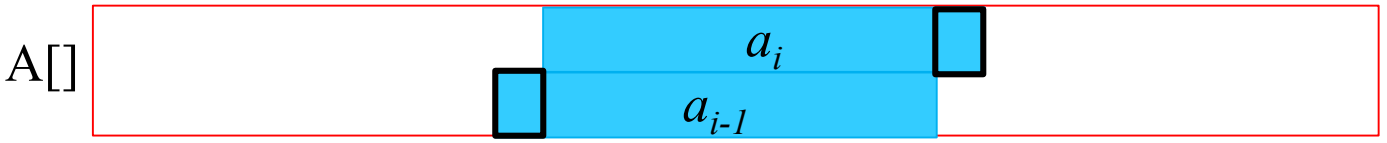
- 문자열 패턴을 수치로 바꾸어 문자열의 비교를 수치 비교로 대신한다
- 수치화
  - 가능한 문자 집합 $\sum$의 크기에 따라 진수가 결정된다
  - 예: $\sum = \{a, b, c, d, e\}$
    - $|\sum| = 5$
    - a, b, c, d, e를 각각 0, 1, 2, 3, 4에 대응시킨다
    - 문자열 "cad"를 수치화하면 $2*5^2+0*5^1+3*5^0 = 28$

수치화 작업의 부담

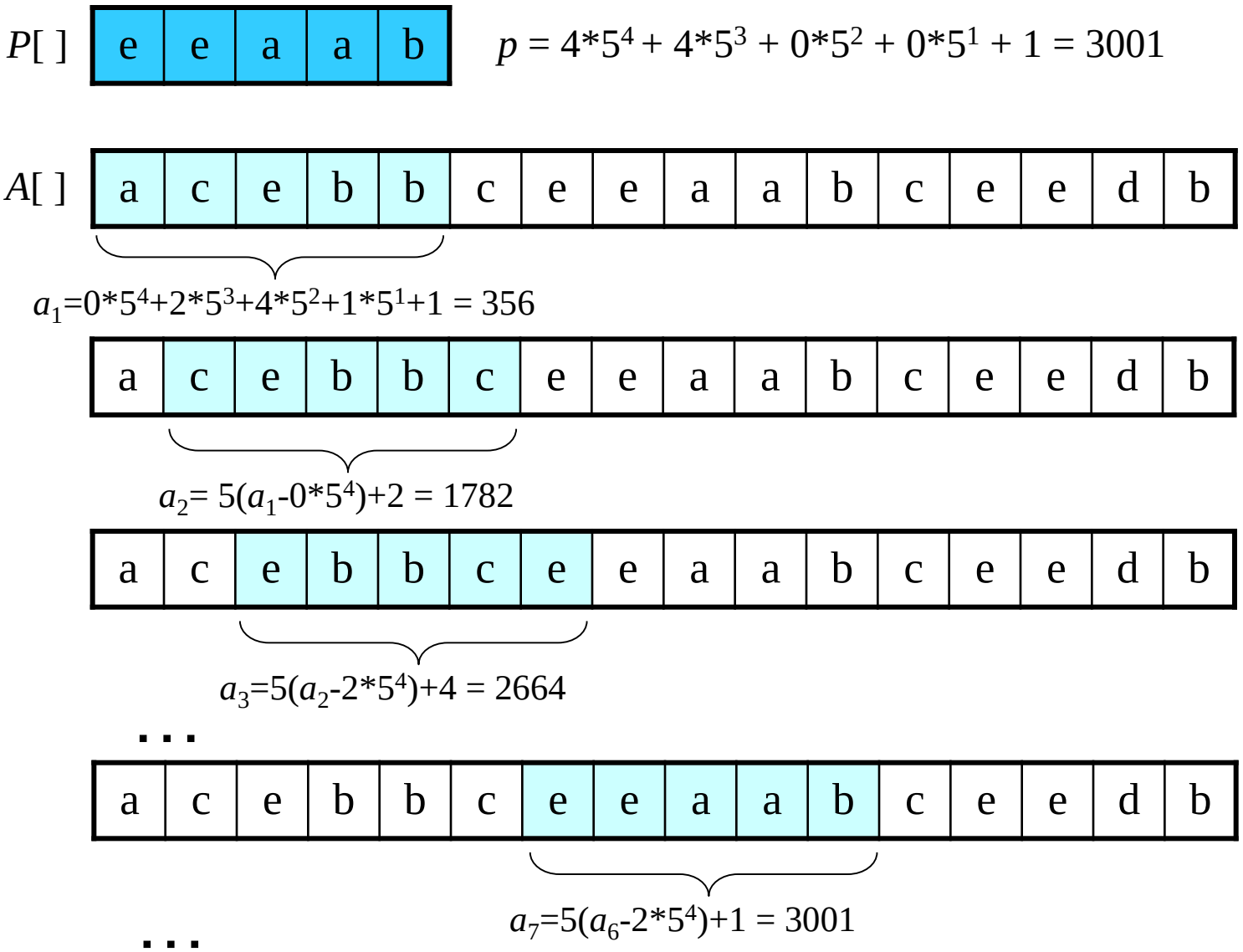
abbafcdabafbeabebacababababacaagb…

- $A[i…i+m-1]$에 대응되는 수치의 계산
  - $a_i = A[i+m-1] + d(A[i+m-2] + d(A[i+m-3] + d(… + d(A[i]))…)$
  - $\Theta(m)$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



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;
    }
}
```

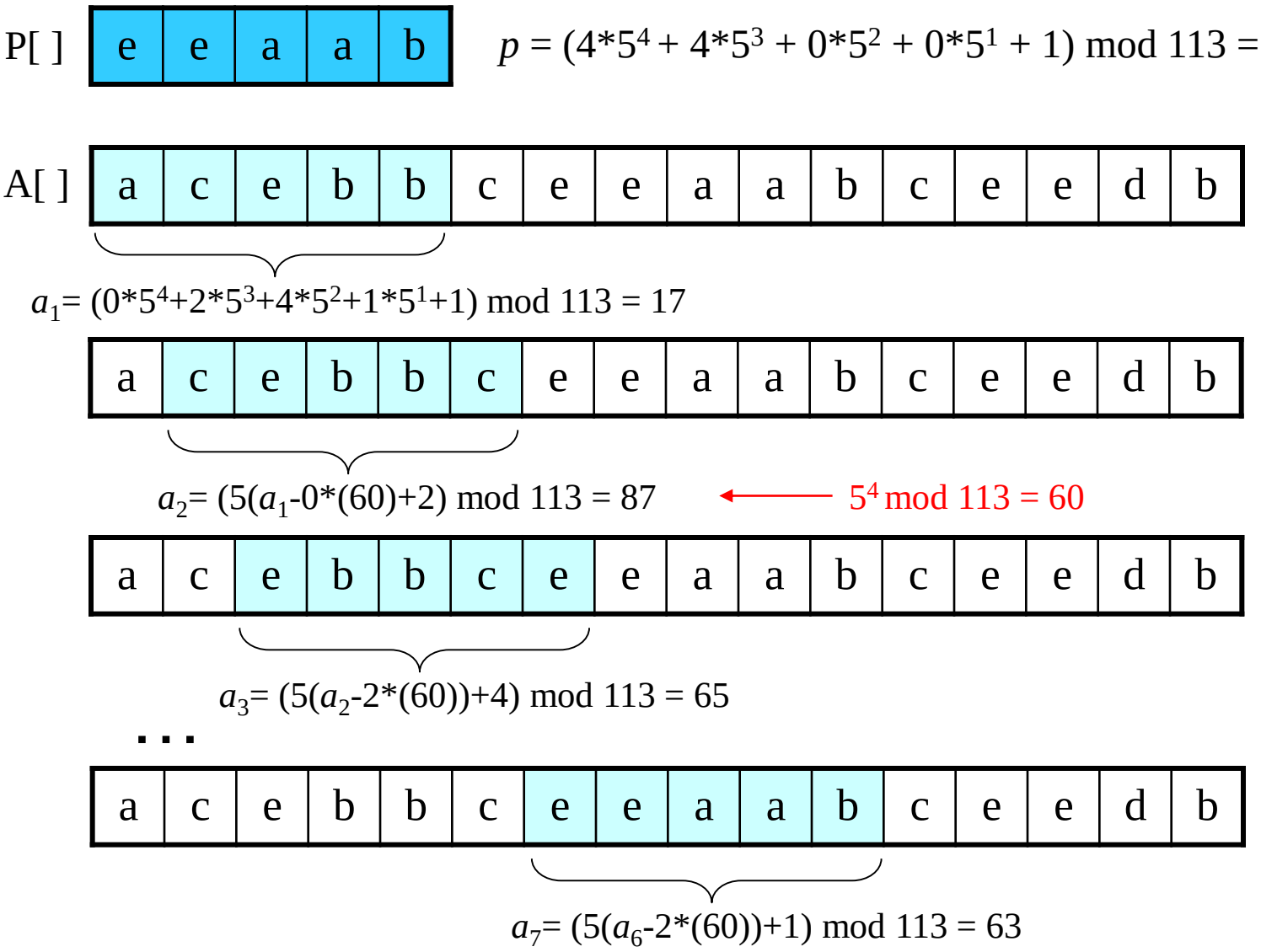
✓ 총 수행시간: $\Theta(n)$

앞의 알고리즘의 문제점

- 문자 집합 Σ와 m의 크기에 따라 a_i가 매우 커질 수 있다
  - 심하면 computer word의 용량 초과
  - overflow 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, dqm이 레지스터에 수용될 수 있도록 잡는다

나머지 연산을 이용한 매칭의 예



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;
    }
}
```

실제 매칭이 아닌데 이 값이 같을 확률: $1/q$

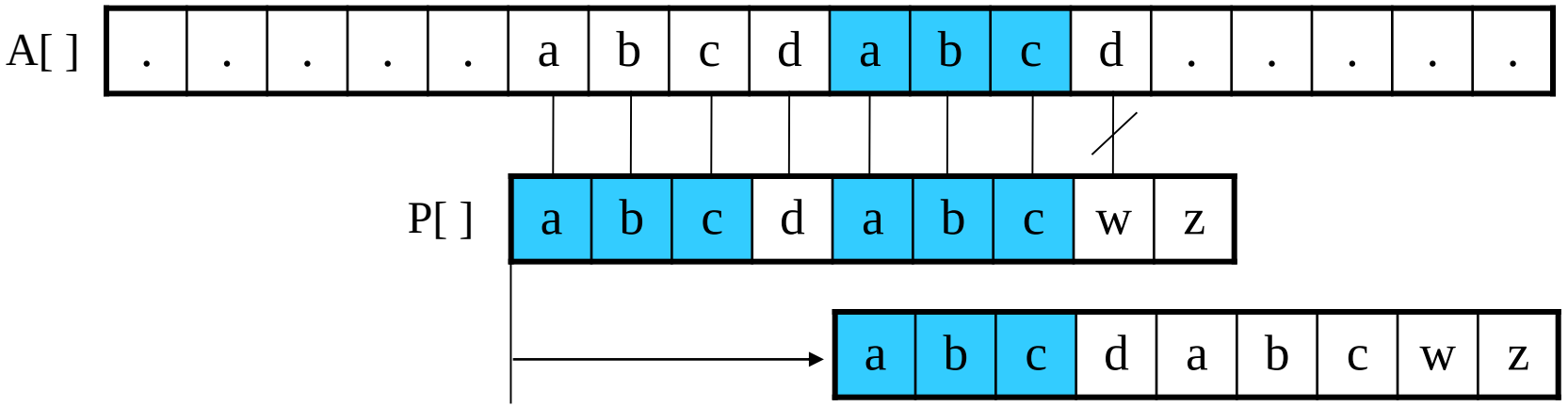
우연히 $p = b_i$가 되는 횟수의 기대치: n/q

보통 $n << q$

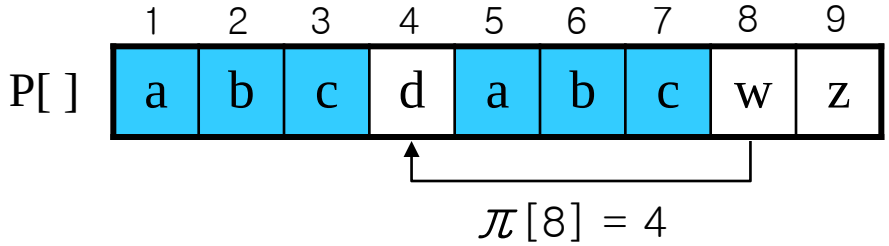
✓ 평균 수행시간: $\Theta(n)$

KMP Knuth-Morris-Pratt 알고리즘

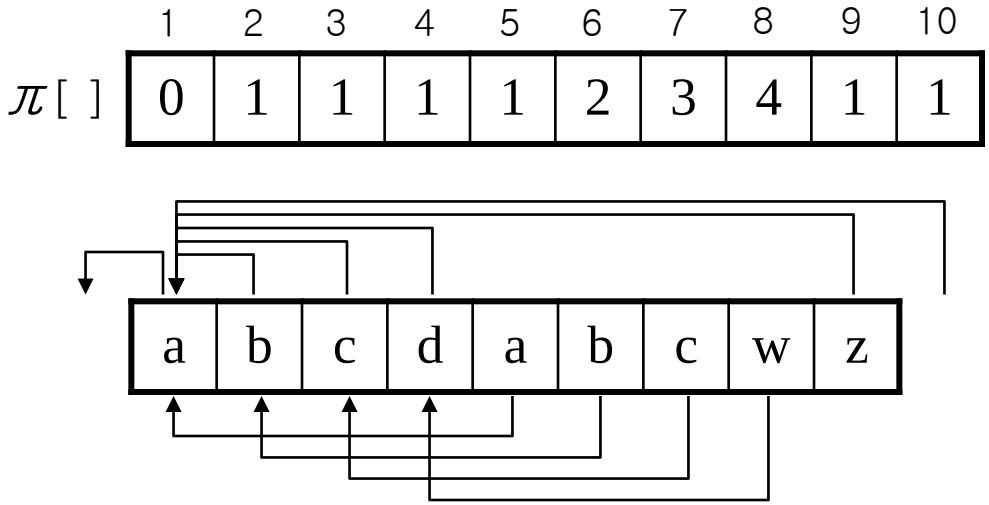
- 오토마타를 이용한 매칭과 동기가 유사
- 공통점
  - 매칭에 실패했을 때 돌아갈 상태를 준비해둔다
  - 오토마타를 이용한 매칭보다 준비 작업이 단순하다



매칭이 실패했을 때 돌아갈 곳 준비 작업



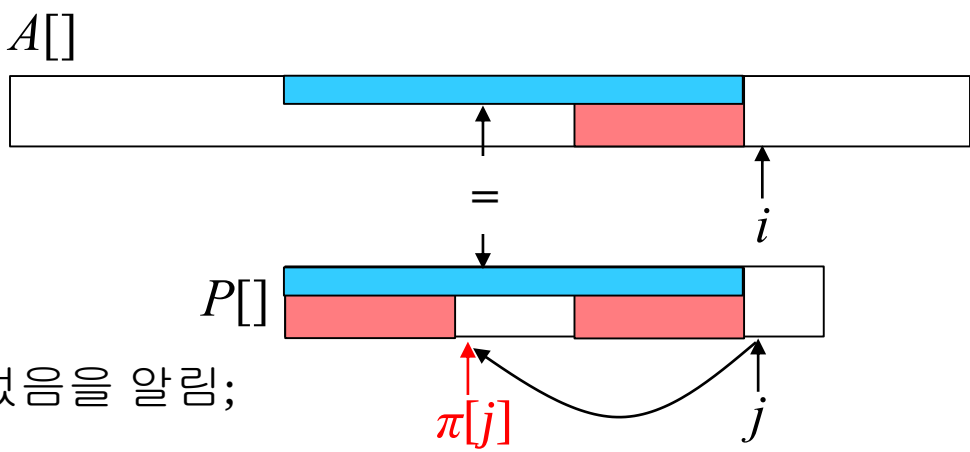
텍스트에서 abcdabc까지는 매치되고, w에서 실패한 상황
패턴의 맨앞의 abc와 실패 직전의 abc는 동일함을 이용할 수 있다
실패한 텍스트 문자와 P[4]를 비교한다



패턴의 각 위치에 대해
매칭에 실패했을 때
돌아갈 곳을 준비해 둔다

KMP 알고리즘

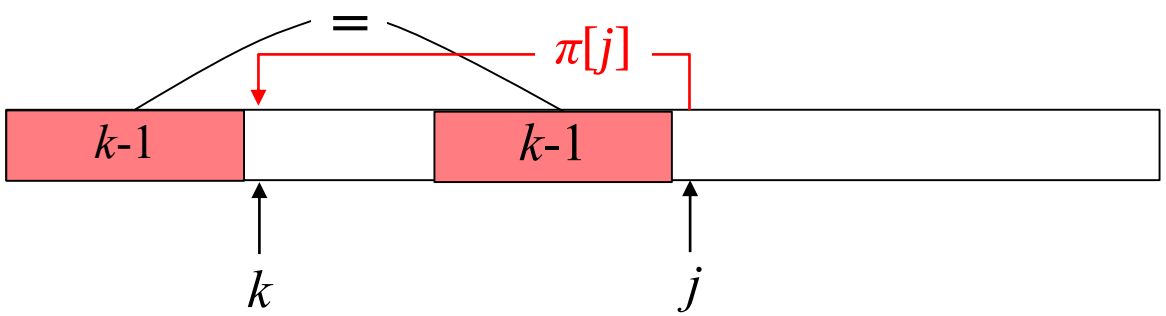
```
KMP(A[ ], P[ ], n, m)
{  ▷ n: 배열 A[ ]의 길이, m: 배열 P[ ]의 길이
    preprocessing(P);
    i ← 1;  ▷ 본문 문자열 포인터
    j ← 1;  ▷ 패턴 문자열 포인터
    while (i ≤ n) {
        if (j = 0 or A[i] = P[j])
            then { i++;  j++; }
            else  j ← π [j];
        if (j = m+1) then {
            A[i-m]에서 매치되었음을 알림;
            j ← π [j];
        }
    }
}
```



✓수행시간: $\Theta(n)$

준비 작업

```
preprocessing(P[ ], m)
{  ▷ m: 배열 P[ ]의 길이
    j ← 1;
    k ← 0;  ▷ prefix 포인터
    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: $\Theta(m)$

KMP의 수행시간 분석

- Every time we go through the loop, the algorithm advances in the text (by i++) or shift the pattern (by $j \leftarrow \pi[j]$).
- Note that $\forall j, \pi[j] < j$, so $j \leftarrow \pi[j]$ decreases j.
- Thus, each time we go through the loop, $i+(i-j)$ will be increased by at least 1.
- $i + (i - j) \leq 2i \leq 2n$, i.e., we go through the loop at most $2n$ times.
- Since each while loop takes $\theta(1)$, the running time is $O(n)$.

Since Ω(n), finally Θ(n)

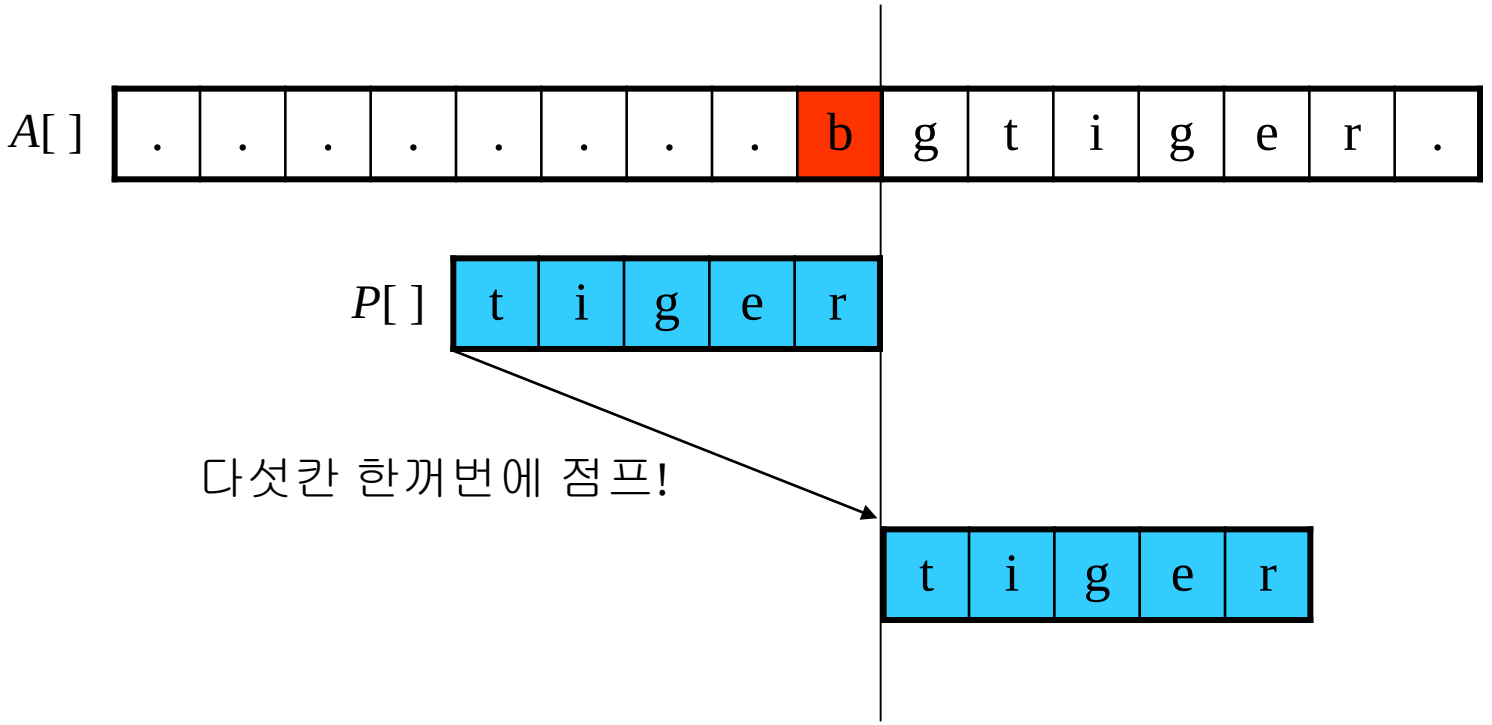
```
while (i ≤ n) {
    if (j = 0 or A[i] = P[j])
        then { i++; j++; }
        else j ← π [j];
    if (j = m+1) then {
        A[i-m]에서 매치되었음을 알림;
        j ← π [j];
    }
}
```

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

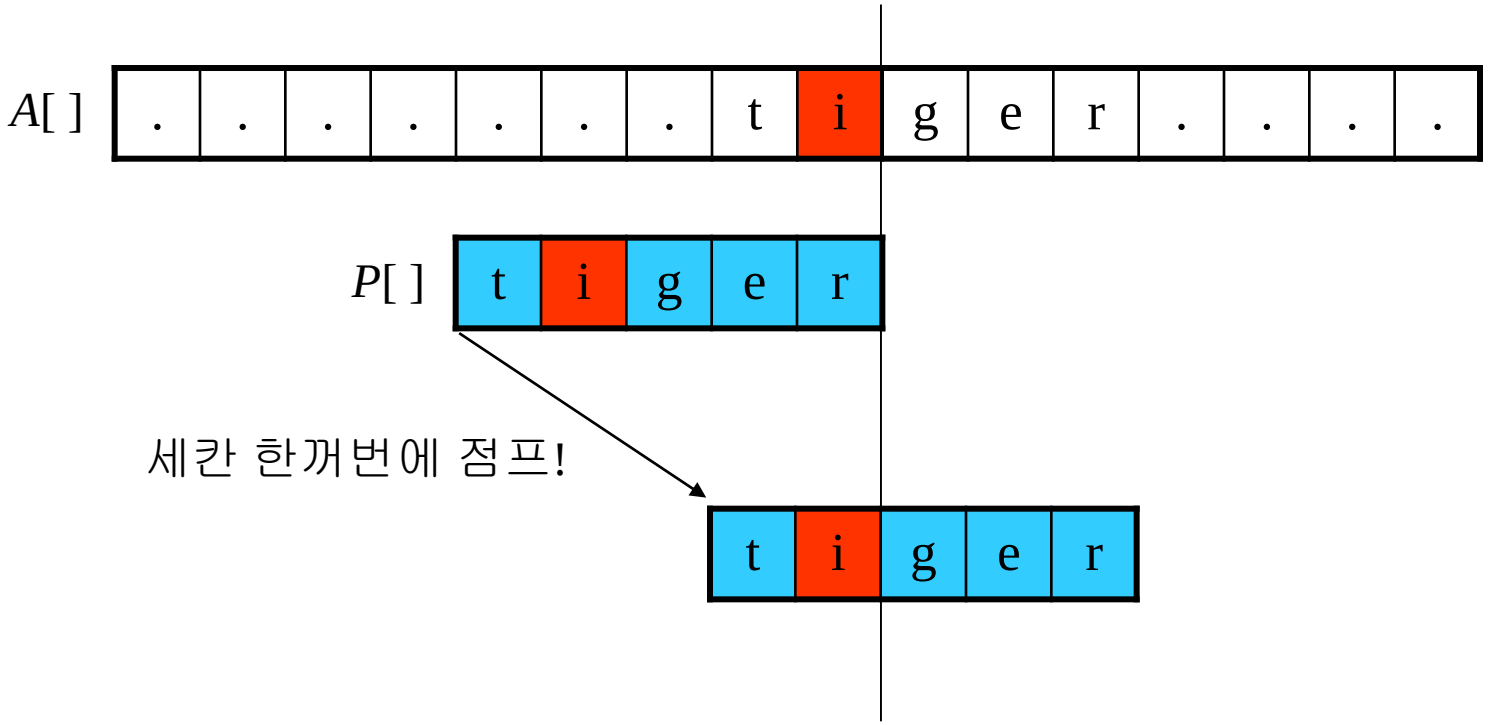
Motivation

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

보이어-무어-호스풀 알고리즘

```
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;
        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]];
    }
}
```

- ✓Worst case: $\Theta(mn)$
- ✓입력에 따라 다르지만 일반적으로 $\Theta(n)$보다 가볍다
- ✓ Best case: $\Theta(\frac{n}{m})$