스트링 매칭

Lecture 7

스트링 매칭

- 주어진 text string 에서 pattern string 이 나타나는 (모든, 선두) 위치를 결정하는 문제
  - 문서내에서 어떤 단어가 어느 곳에 나타나는지 찾아내는 경우
  - 패턴인식, 음석인식, DNA 염기 서열 분석 등에 응용

- Notation
  - 스트링(string, sequence): 문자가 연속적으로 나열된 것
  - 알파벳 Σ : 스트링에 나타날 수 있는 문자들 집합
  - Text string: 길이가 n인 배열, T[0..n-1]
  - Pattern string: 길이가 m 인 배열 P[0..m-1], m<n
  - 공 스트링 = ε

스트링 매칭

- 두 스트링 x, y의 접속 = xyEx. x=abc, y=def 이면 xy=abcdef
- x=wy
  - w : x의 접두부(prefix) $w \sqsubset x$ (c.f. 비교대상 string을 뒤에 표기)
  - y : x의 접미부(suffix) $y \sqsupset x$
  - $w \sqsubset x$ 이면서 w≠x인 경우 w는 x의 진접두부 (proper prefix)
  - $y \sqsupset x$ 이면서 y≠x인 경우 y는 x의 진접미부 (proper suffix)

스트링 매칭

- 예
  - ab $\sqsubset$ abcde
  - de $\sqsupset$ abcde
  - 공 스트링 = ε은 모든 스트링의 prefix 이면서 suffix
  - 두 스트링 x, y와 임의의 문자 a에 대하여 $y \sqsupset x$ 이면 $ya \sqsupset xa$, 역도 성립
  - 추이적 관계(transitive relation): u가 v의 prefix이고 v가 w의 suffix 이면 u 는 w의 접두부임

직선적 알고리즘

- 주어진 text string T: a b a c a b a b a b c a_b a b a b b a c a a c...

  pattern string P: a b a b a b c a

  P: a b a b a b c a

  P: a b a b a b c a

  P: a b a b a b c a

  P: a b a b a b c a

- 텍스트의 각 위치에서 시작하는 부분 스트링이 패턴과 일치하는지 여부를 조사하는 방법

직선적 알고리즘

T: a b a c a b a b a b c a b a b a b b a c a a c...

P: a b a b a b c a

```
BruteForce(T, P, n, m)
/*  입력 : T : 텍스트, 크기가 n인 문자의 배열
           P : 패턴, 크기가 m인 문자의 배열
   출력 : 텍스트 내의 패턴이 존재하는 위치 */
 {
       for (i = 0; i ≤ n-m; i + + ) {
           for (j = 0; j < m; j + + ) {
               if ( P[j] ! = T[i + j] ) break ;
           }
           if (j = = m) printf("패턴이 텍스트의 i번째부터 나타남") ;
    }
 }
```

직선적 알고리즘

- 최악의 경우,
  m(n-m+1) 비교 후 첫 번째 매칭 발견

  T: 0 0 0 0 0 ...... 0 0 0 0 0 1

  P: 0 0 0 0 1

  $O(m(n-m+1))=O(mn)$

라빈 카프 알고리즘
Rabin-Karp Algorithm

- 스트링을 숫자 값으로 바꾼 뒤 해시(hash)값을 계산해 매칭
  Ex. 영문알파벳 26글자 > 26진법의 숫자 값으로 변환 가능

- 10진수로만 이루어진 스트링 예시
  - Σ= {0, 1, … , 9}

    T: 25436712345678

    P: 1234
  - 패턴과 텍스트의 각 원소는 실제로는 문자이지만 패턴 전체를 하나의 십진수로 바꾸어 생각함.
  - P 전체가 하나의 숫자가 되고 텍스트는 각 위치마다 길이 m만큼의 10진수인것으로 생각하여 이 숫자값을 비교함.

라빈 카프 알고리즘

T: 25436712345678

P: 1234

  P: 1234

    P: 1234

- P 전체가 하나의 숫자가 되고 텍스트는 각 위치마다 길이 m만큼의 10진수인것으로 생각하여 이 숫자값을 비교함.
- P를 한 워드로 나타낼 경우 n-m+1번의 비교로 매칭 가능.

라빈 카프 알고리즘

- 패턴 P[0..m-1]에 대한 10진수 p의 계산은 호너(Horner)의 방법을 써서 O(m)시간에 계산할 수 있음.

$$p = P[0]10^{m-1} + P[1]10^{m-2} + \ldots + P[m-1]10^{0}$$

- 패턴 T[0..m-1]에 대한 10진수 t의 계산 역시 호너(Horner)의 방법을 써서 O(m)시간에 계산할 수 있음.

$$t_0 = T[0]10^{m-1} + T[1]10^{m-2} + \ldots + T[m-1]10^{0}$$

$$t_s = T[s]10^{m-1} + T[s+1]10^{m-2} + \ldots + T[s+m-1]10^{0}$$

- 예) 2543, m=4

$2543 = 2 \times 10^3 + 5 \times 10^2 + 4 \times 10 + 3$

라빈 카프 알고리즘

- t_s와 t_{s+1} 사이 관계를 이용한 점화식 통해 반복적인 계산 과정 배제 가능,10^{m-1}만 미리 계산해 놓는다면 t_s로 부터 t_{s+1}을 상수 시간에 계산 할 수 있음.

$$t_{s+1} = 10(t_s - 10^{m-1}T[s]) + T[s+m]$$

- 예) 25436, m=4, $10(25436-2\times10^3)+6=5436$
- t_0 를 O(m)시간에 계산 할 수 있고 이로부터 나머지 $t_1, t_2, \ldots, t_{n-m}$ 을 O(n-m)시간에 계산 할 수 있다.

라빈 카프 알고리즘

- 일반적으로 알파벳의 크기가 d라면 알파벳의 각 문자를 0~d-1까지의 숫자에 대응시켜 매칭을 해도 마찬가지임.
- Σ= {0, 1, … , d-1}인 d진법의 알파벳이라면

$$p = P[0]d^{m-1} + P[1]d^{m-2} + \ldots + P[m-1]d^{0}$$

$$t_0 = T[0]d^{m-1} + T[1]d^{m-2} + \ldots + T[m-1]d^{0}$$

$$t_s = T[s]d^{m-1} + T[s+1]d^{m-2} + \ldots + T[s+m-1]d^{0}$$

$$t_{s+1} = d(t_s - d^{m-1}T[s]) + T[s+m]$$

라빈 카프 알고리즘

- p나 t_s가 m개의 문자를 포함할때 이를 이대로 계산하면 이 값이 매우 크게 될 가능성이 많음.

p안에서의 각 산술 연산(X, +)이 상수시간에 수행이 불가능함

- String으로 hash 값을 계산한 후 동일(또는 유사) 여부를 판단하여 매칭

해시(Hash): 하나의 문자열을, 이를 상징하는 더 짧은 길이의 값이나 키로 변환하는 것이다

- Hash 함수: h(p) = p mod q
단, q는 d*q가 한 워드로 되는 최대 소수(prime number)를 사용해 필요한 산술연산이 단일-크기의 산술 연산이 되도록 함.

$$h = p \bmod q$$

$$\tau_0 = t_0 \bmod q$$

$$\tau_{s+1} = (d(\tau_s - d^{m-1}T[s]) + T[s+m]) \bmod q$$

라빈 카프 알고리즘

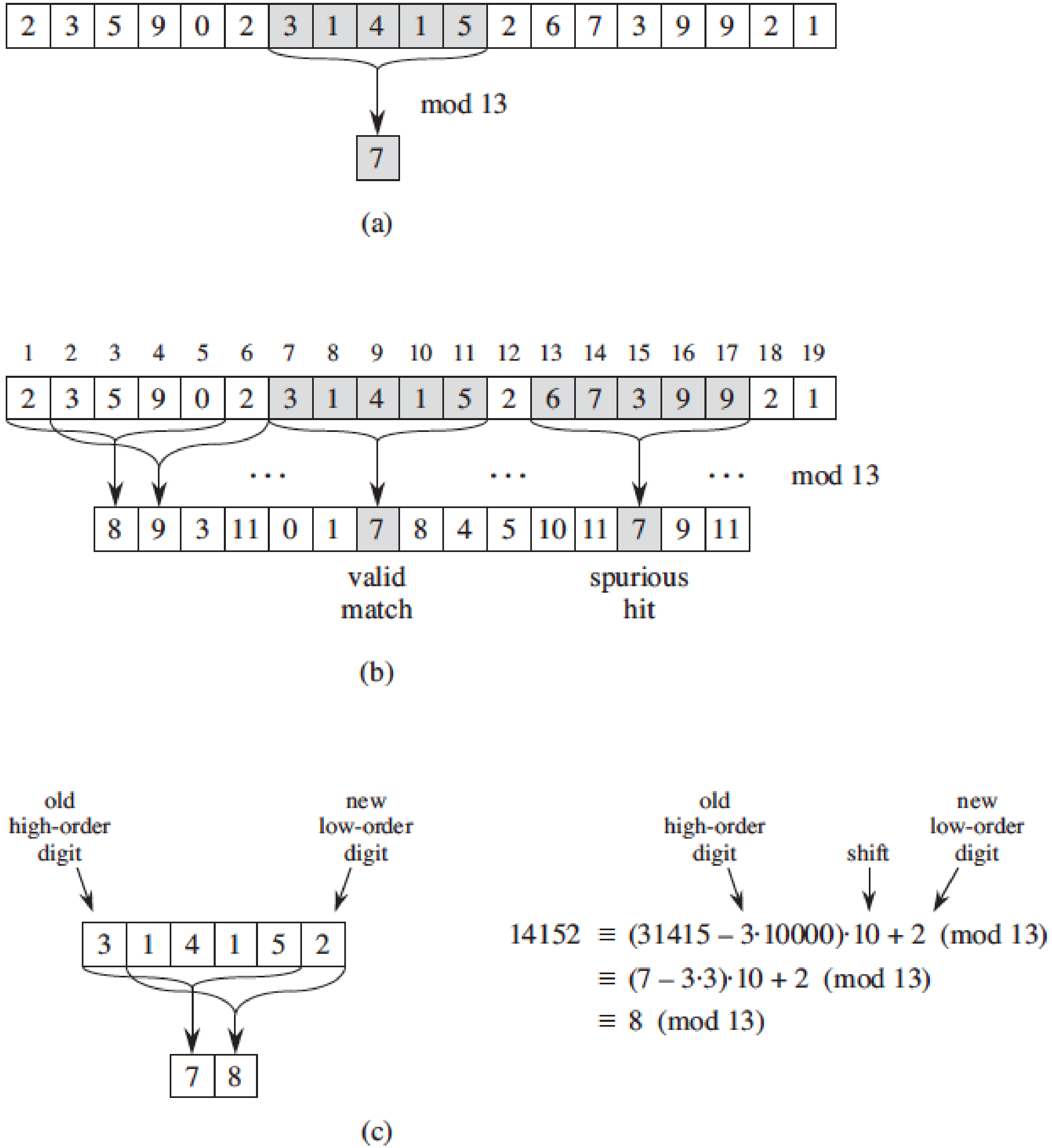
$$\tau_{s+1} = (d(\tau_s - d^{m-1}T[s]) + T[s+m]) \bmod q$$

- 이대로 계산하면 계산과정에서 큰수가 나타날 수 있음.
- mod 연산의 성질 (곱셈, 덧셈, 뺄셈에 모두 성립)

$$(a \cdot b) \bmod q = ((a \bmod q)(b \bmod q)) \bmod q$$
$$= ((a \bmod q \cdot b) \bmod q$$
$$= (a \cdot (b \bmod q)) \bmod q$$

- mod 연산을 통한 hash 값 계산으로 t_{s+1}의 빠른 계산 가능 (d^{m-1} 값을 사전 계산할 경우 상수 시간 내 계산 가능)

$$\tau_{s+1} = (d(\tau_s - d^{m-1}T[s]) + T[s+m]) \bmod q$$
$$= (d(\tau_s - (d^{m-1} \bmod q)T[s]) + T[s+m]) \bmod q$$
$$= (d(\tau_s - DT[s]) + T[s+m]) \bmod q$$


2 3 5 9 0 2 3 1 4 1 5 2 6 7 3 9 9 2 1
mod 13
7
(a)
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
2 3 5 9 0 2 3 1 4 1 5 2 6 7 3 9 9 2 1
mod 13
8 9 3 11 0 1 7 8 4 5 10 11 7 9 11
valid match
spurious hit
(b)
old high-order digit
new low-order digit
3 1 4 1 5 2
7 8
old high-order digit
shift
new low-order digit
14152 ≡ (31415 – 3·10000)·10 + 2 (mod 13)
≡ (7 – 3·3)·10 + 2 (mod 13)
≡ 8 (mod 13)
(c)

라빈 카프 알고리즘

- $t_s \not\equiv p \pmod{q}$ 이면 $t_s \neq p$ 이므로 부적절한 시프트 s를 제거해줌
- $t_s \equiv p \pmod{q}$ 의 해가 $t_s = p$ 를 의미하지 않으므로 모듈로 q 를 이용해 구한 해는 완벽하지 않음.
- $t_s \equiv p \pmod{q}$ 경우에는 임의의시프트 s를 좀 더 조사해 이 값이 타당한지 가짜 적중인지 조사해야 함. 이는 P[1..m]=ㅅT[s+1..s+m]을 조사해 알 수 있다. q 가 크면 가짜 적중이 발생하는 경우가 다주 드물기 때문에 추가 검사 비용이 매우 저렴하다.

라빈 카프 알고리즘

```
Rabin-Karp(T,P,d,q){
/* 입력 : T : 텍스트, 크기가 n인 문자의 배열 T[0..n-1]
d : 알파벳의 크기        q : 해시 함수에 의해 결정
        D = d^(m-1) mod q;                              //d^(m-1) 값 사전계산
          h = 0;  t = 0;
          for (i = 0; i ≤ m-1; i++) {                   //Horner(호너) 방법을 통한 h, t 계산
                h = (d*h + P[i]) mod q;
                t = (d*t + T[i]) mod q;
        }
         for (s = 0; s <  n-m+1; s++) {                  //텍스트 각 위치 별로 패턴 매칭
             if ( h == t ) {                             //hash 값이 동일한 경우에 한해 세부 패턴 매칭 진행
               for (i = 0; i < m; i++)
                  if (P[i] != T[s+i]) break;
                  if (i == m)
                      printf ("패턴이 위치 s에서 발생"); }
             if (s < n-m) t = (d*(t-T[s]*D) + T[s+m]) mod q; //t_s와 t_(s+1) 사이 관계를 이용한 점화식 통해
다음 t값 계산
          }
}
```

라빈 카프 알고리즘

- 최악의 경우: O((n-m+1)m) = O(mn)

모든 위치에서 해시 값이 패턴의 해시 값과 일치. 그때마다 텍스트의 문자와 패턴의 문자를 하나하나 전부 비교해야함 불일치하는

- 최선의 경우: O(m+n)

대부분의 경우 텍스트의 해시값과 패턴의 해시 값이 일치하지 않음. 그 경우 패턴과 텍스트 문자의 비교가 필요하지 않음.

따라서 실제로 대부분의 O(m+n)시간에 실행될 가능성이 높음

KMP 알고리즘

- Knuth, Morris, Pratt 세사람이 고안한 알고리즘
- 직선적 방법 : 최악의 경우 O(mn)

T: a d c b a d e a d c b a d c b a d c f

P: a d c b a d c f

a d c b a d c f

a d c b a d c f

a d c b a d c f

a d c b a d c f

…

a d c b a d c f

KMP 알고리즘

- suffix/prefix match 를 이용해 불필요한 비교 반복을 없앰

T: a d c b a d e a d c b a d c b a d c f

P: a d c b a d c f

KMP 알고리즘

- suffix/prefix match 를 이용해 불필요한 비교 반복을 없앰

T: a d c b a d e a d c b a d c b a d c f

P: a d c b a d c f

prefix == suffix

KMP 알고리즘

- suffix/prefix match 를 이용해 불필요한 비교 반복을 없앰

T: a d c b a d e a d c b a d c b a d c f

P: a d c b a d c f

prefix == suffix

KMP 알고리즘

- suffix/prefix match 를 이용해 불필요한 비교 반복을 없앰

T: a d c b a d e a d c b a d c b a d c f

a d c b a d c f

KMP 알고리즘

- suffix/prefix match 를 이용해 불필요한 비교 반복을 없앰

T: a d c b a d e a d c b a d c b a d c f

a d c b a d c f

no suffix prefix match

KMP 알고리즘

- suffix/prefix match 를 이용해 불필요한 비교 반복을 없앰

T: a d c b a d e a d c b a d c b a d c f

a d c b a d c f

KMP 알고리즘

- suffix/prefix match 를 이용해 불필요한 비교 반복을 없앰

T: a d c b a d e a d c b a d c b a d c f

a d c b a d c f

KMP 알고리즘

- suffix/prefix match 를 이용해 불필요한 비교 반복을 없앰

T: a d c b a d e a d c b a d c b a d c f

a d c b a d c f

prefix == suffix

KMP 알고리즘

- suffix/prefix match 를 이용해 불필요한 비교 반복을 없앰

T: a d c b a d e a d c b a d c b a d c f

a d c b a d c f

prefix == suffix

KMP 알고리즘

- suffix/prefix match 를 이용해 불필요한 비교 반복을 없앰

T: a d c b a d e a d c b a d c b a d c f

a d c b a d c f

KMP 알고리즘

- suffix/prefix match 를 이용해 불필요한 비교 반복을 없앰

T: a d c b a d e a d c b a d c b a d c f

a d c b a d c f

KMP 알고리즘

- suffix/prefix match 를 이용해 불필요한 비교 반복을 없앰

T: a d c b a d e a d c b a d c b a d c f

a d c b a d c f

KMP 알고리즘

- suffix/prefix match 를 이용해 불필요한 비교 반복을 없앰

T: a d c b a d e a d c b a d c b a d c f

a d c b a d c f

KMP 알고리즘

- suffix/prefix match 를 이용해 불필요한 비교 반복을 없앰

T: a d c b a d e a d c b a d c b a d c f

a d c b a d c f

KMP 알고리즘

- suffix/prefix match 를 이용해 불필요한 비교 반복을 없앰

T: a d c b a d e a d c b a d c b a d c f

a d c b a d c f

KMP 알고리즘

- suffix/prefix match 를 이용해 불필요한 비교 반복을 없앰

T: a d c b a d e a d c b a d c b a d c f

a d c b a d c f

KMP 알고리즘

- suffix/prefix match 를 이용해 불필요한 비교 반복을 없앰

T: a d c b a d e a d c b a d c b a d c f

a d c b a d c f

KMP 알고리즘

- suffix/prefix match 를 이용해 불필요한 비교 반복을 없앰

T: a d c b a d e a d c b a d c b a d c f

a d c b a d c f

KMP 알고리즘

- suffix/prefix match 를 이용해 불필요한 비교 반복을 없앰

T: a d c b a d e a d c b a d c b a d c f

a d c b a d c f

no suffix prefix match

KMP 알고리즘

- suffix/prefix match 를 이용해 불필요한 비교 반복을 없앰

T: a d c b a d e a d c b a d c b a d c f

a d c b a d c f

KMP 알고리즘

- suffix/prefix match 를 이용해 불필요한 비교 반복을 없앰

T: a d c b a d e a d c b a d c b a d c f

a d c b a d c f

KMP 알고리즘

- suffix/prefix match 를 이용해 불필요한 비교 반복을 없앰

T: a d c b a d e a d c b a d c b a d c f

a d c b a d c f

KMP 알고리즘

- suffix/prefix match 를 이용해 불필요한 비교 반복을 없앰

T: a d c b a d e a d c b a d c b a d c f

a d c b a d c f

KMP 알고리즘

- suffix/prefix match 를 이용해 불필요한 비교 반복을 없앰

T: a d c b a d e a d c b a d c b a d c f

a d c b a d c f

KMP 알고리즘

- suffix/prefix match 를 이용해 불필요한 비교 반복을 없앰

T: a d c b a d e a d c b a d c b a d c f

a d c b a d c f

KMP 알고리즘

- suffix/prefix match 를 이용해 불필요한 비교 반복을 없앰

T: a d c b a d e a d c b a d c b a d c f

a d c b a d c f

KMP 알고리즘

- suffix/prefix match 를 이용해 불필요한 비교 반복을 없앰

T: a d c b a d e a d c b a d c b a d c f

a d c b a d c f

KMP 알고리즘

- suffix/prefix match 를 이용해 불필요한 비교 반복을 없앰

T: a d c b a d e a d c b a d c b a d c f

a d c b a d c f

KMP 알고리즘

- suffix/prefix match 를 이용해 불필요한 비교 반복을 없앰

T: a d c b a d e a d c b a d c b a d c f

a d c b a d c f

KMP 알고리즘

- suffix/prefix match 를 이용해 불필요한 비교 반복을 없앰

T: a d c b a d e a d c b a d c b a d c f

a d c b a d c f

KMP 알고리즘

- suffix/prefix match 를 이용해 불필요한 비교 반복을 없앰

T: a d c b a d e a d c b a d c b a d c f

a d c b a d c f

KMP 알고리즘

- suffix/prefix match 를 이용해 불필요한 비교 반복을 없앰

T: a d c b a d e a d c b a d c b a d c f

a d c b a d c f

KMP 알고리즘

- suffix/prefix match 를 이용해 불필요한 비교 반복을 없앰

T: a d c b a d e a d c b a d c b a d c f

a d c b a d c f

KMP 알고리즘

- 매칭 전에 일단 Suffix 와 Prefix 가 어디서 일치하는지 알아야함 -> 최대 접두부 테이블 만들기

P	a	c	b	d	a	c	b	a
index	0	1	2	3	4	5	6	7
j								
i								
SP	0							

KMP 알고리즘

- 매칭 전에 일단 Suffix 와 Prefix 가 어디서 일치하는지 알아야함 -> 최대 접두부 테이블 만들기

P	a	c	b	d	a	c	b	a
index	0	1	2	3	4	5	6	7
j	j							
i		i						
SP	0	0						

SP[j] != SP[i]

KMP 알고리즘

- 매칭 전에 일단 Suffix 와 Prefix 가 어디서 일치하는지 알아야함 -> 최대 접두부 테이블 만들기

P	a	c	b	d	a	c	b	a
index	0	1	2	3	4	5	6	7
j	j							
i			i					
SP	0	0	0					

SP[j] != SP[i]

KMP 알고리즘

- 매칭 전에 일단 Suffix 와 Prefix 가 어디서 일치하는지 알아야함 -> 최대 접두부 테이블 만들기

P	a	c	b	d	a	c	b	a
index	0	1	2	3	4	5	6	7
j	j							
i				i				
SP	0	0	0	0				

SP[j] != SP[i]

KMP 알고리즘

- 매칭 전에 일단 Suffix 와 Prefix 가 어디서 일치하는지 알아야함 -> 최대 접두부 테이블 만들기

P	a	c	b	d	a	c	b	a
index	0	1	2	3	4	5	6	7
j	j							
i					i			
SP	0	0	0	0	1			

SP[j] == SP[i]

KMP 알고리즘

- 매칭 전에 일단 Suffix 와 Prefix 가 어디서 일치하는지 알아야함 -> 최대 접두부 테이블 만들기

P	a	c	b	d	a	c	b	a
index	0	1	2	3	4	5	6	7
j		j						
i						i		
SP	0	0	0	0	1			

SP[j] == SP[i]
SP[i]=j+1

KMP 알고리즘

- 매칭 전에 일단 Suffix 와 Prefix 가 어디서 일치하는지 알아야함 -> 최대 접두부 테이블 만들기

P	a	c	b	d	a	c	b	a
index	0	1	2	3	4	5	6	7
j		j						
i						i		
SP	0	0	0	0	1			

1의 의미? 길이 1인 a는 이미 비교되었으므로 다음은 f와 c 비교하면 된다는 뜻.

T:acbdaf

P:acbdac

KMP 알고리즘

- 매칭 전에 일단 Suffix 와 Prefix 가 어디서 일치하는지 알아야함 -> 최대 접두부 테이블 만들기

P	a	c	b	d	a	c	b	a
index	0	1	2	3	4	5	6	7
j		j						
i						i		
SP	0	0	0	0	1	2		

SP[j] == SP[i]
SP[i]=j+1

KMP 알고리즘

- 매칭 전에 일단 Suffix 와 Prefix 가 어디서 일치하는지 알아야함 -> 최대 접두부 테이블 만들기

P	a	c	b	d	a	c	b	a
index	0	1	2	3	4	5	6	7
j			j					
i							i	
SP	0	0	0	0	1	2	3	

SP[j] == SP[i]
SP[i]=j+1

KMP 알고리즘

- 매칭 전에 일단 Suffix 와 Prefix 가 어디서 일치하는지 알아야함 -> 최대 접두부 테이블 만들기

P	a	c	b	d	a	c	b	a
index	0	1	2	3	4	5	6	7
j				j				
i								i
SP	0	0	0	0	1	2	3	

SP[j] != SP[i]

KMP 알고리즘

- 매칭 전에 일단 Suffix 와 Prefix 가 어디서 일치하는지 알아야함 -> 최대 접두부 테이블 만들기

P	a	c	b	d	a	c	b	a
index	0	1	2	3	4	5	6	7
j				j				
i								i
SP	0	0	0	0	1	2	3	

SP[j] != SP[i]

KMP 알고리즘

- 매칭 전에 일단 Suffix 와 Prefix 가 어디서 일치하는지 알아야함 -> 최대 접두부 테이블 만들기

P	a	c	b	d	a	c	b	a
index	0	1	2	3	4	5	6	7
j	j			j				
i								i
SP	0	0	0	0	1	2	3	1

SP[j] == SP[i]
SP[i]=j+1

KMP 알고리즘

- 최대 접두부 테이블 만들기

a	c	a	c	a	b	a	c	a	c	a	b	a	c	a	c	a	c
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
j																	
	i																
0																	

KMP 알고리즘

- 최대 접두부 테이블 만들기

a	c	a	c	a	b	a	c	a	c	a	b	a	c	a	c	a	c
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
j																	
	i																
0	0																

KMP 알고리즘

- 최대 접두부 테이블 만들기

a	c	a	c	a	b	a	c	a	c	a	b	a	c	a	c	a	c
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
j																	
		i															
0	0	1															

KMP 알고리즘

- 최대 접두부 테이블 만들기

a	c	a	c	a	b	a	c	a	c	a	b	a	c	a	c	a	c
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
	j																
			i														
0	0	1	2														

KMP 알고리즘

- 최대 접두부 테이블 만들기

a	c	a	c	a	b	a	c	a	c	a	b	a	c	a	c	a	c
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
		j															
				i													
0	0	1	2	3													

KMP 알고리즘

- 최대 접두부 테이블 만들기

a	c	a	c	a	b	a	c	a	c	a	b	a	c	a	c	a	c
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
			j														
					i												
0	0	1	2	3													

SP[j] != SP[i]
i랑 비교할 다음 j는 누구?

KMP 알고리즘

- 최대 접두부 테이블 만들기

a c a c a b a c a c a b a c a c a c

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

j j

i

0 0 1 2 3

SP[j] != SP[i]
i랑 비교할 다음 j는 누구? c

KMP 알고리즘

- 최대 접두부 테이블 만들기

a	c	a	c	a	b	a	c	a	c	a	b	a	c	a	c	a	c
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

j ← j

i

0 0 1 2 3

SP[j] != SP[i]
i랑 비교할 다음 j는 누구?

KMP 알고리즘

- 최대 접두부 테이블 만들기

a	c	a	c	a	b	a	c	a	c	a	b	a	c	a	c	a	c
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
j																	
					i												
0	0	1	2	3	0												

SP[j] != SP[i]
i랑 비교할 다음 j는 누구? a

KMP 알고리즘

- 최대 접두부 테이블 만들기

a	c	a	c	a	b	a	c	a	c	a	b	a	c	a	c	a	c
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
j																	
						i											
0	0	1	2	3	0	1											

KMP 알고리즘

- 최대 접두부 테이블 만들기

a	c	a	c	a	b	a	c	a	c	a	b	a	c	a	c	a	c
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
	j																
							i										
0	0	1	2	3	0	1	2										

KMP 알고리즘

- 최대 접두부 테이블 만들기

a	c	a	c	a	b	a	c	a	c	a	b	a	c	a	c	a	c
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
		j															
								i									
0	0	1	2	3	0	1	2	3									

KMP 알고리즘

- 최대 접두부 테이블 만들기

a	c	a	c	a	b	a	c	a	c	a	b	a	c	a	c	a	c
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
			j														
									i								
0	0	1	2	3	0	1	2	3	4								

KMP 알고리즘

- 최대 접두부 테이블 만들기

a	c	a	c	a	b	a	c	a	c	a	b	a	c	a	c	a	c
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
				j													
										i							
0	0	1	2	3	0	1	2	3	4	5							

KMP 알고리즘

- 최대 접두부 테이블 만들기

a	c	a	c	a	b	a	c	a	c	a	b	a	c	a	c	a	c
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
					j												
											i						
0	0	1	2	3	0	1	2	3	4	5	6						

KMP 알고리즘

- 최대 접두부 테이블 만들기

a	c	a	c	a	b	a	c	a	c	a	b	a	c	a	c	a	c
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
						j											
												i					
0	0	1	2	3	0	1	2	3	4	5	6	7					

KMP 알고리즘

- 최대 접두부 테이블 만들기

a	c	a	c	a	b	a	c	a	c	a	b	a	c	a	c	a	c
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
							j										
													i				
0	0	1	2	3	0	1	2	3	4	5	6	7	8				

KMP 알고리즘

- 최대 접두부 테이블 만들기

a c a c a b a c a c a b a c a c a c

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

j

i

0 0 1 2 3 0 1 2 3 4 5 6 7 8 9

KMP 알고리즘

- 최대 접두부 테이블 만들기

a	c	a	c	a	b	a	c	a	c	a	b	a	c	a	c	a	c
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
									j								
															i		
0	0	1	2	3	0	1	2	3	4	5	6	7	8	9	10		

KMP 알고리즘

- 최대 접두부 테이블 만들기

a c a c a b a c a c a b a c a c a c

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

j

i

0 0 1 2 3 0 1 2 3 4 5 6 7 8 9 10 11

KMP 알고리즘

- 최대 접두부 테이블 만들기

a	c	a	c	a	b	a	c	a	c	a	b	a	c	a	c	a	c
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
											j						
																	i
0	0	1	2	3	0	1	2	3	4	5	6	7	8	9	10	11	

KMP 알고리즘

- 최대 접두부 테이블 만들기

a c a c a b a c a c a b a c a c a c

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

j j

i

0 0 1 2 3 0 1 2 3 4 5 6 7 8 9 10 11

KMP 알고리즘

- 최대 접두부 테이블 만들기

a c a c a b a c a c a b a c a c a c

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

j j

i

0 0 1 2 3 0 1 2 3 4 5 6 7 8 9 10 11 4

j+1

KMP 알고리즘

```
ComputeSP(P, SP, m)
/* 입력: 패턴 P[0..m - 1]
   출력: 최대 접두부 테이블 SP[0..m - 1] */
  {
  1        SP[0] = - 1;
  2        k = - 1;
  3        for ( j = 1; j ≤ m-1; j + + ) {
  4            while ( k ≥ 0 and P[k + 1]≠P[j] ) k = SP[k];
  5            if ( P[k + 1] = = P[j] ) k + + ;
  6            SP[j] = k;
  7        }
    }
```

j	1	2	3	4	5	6	7
P[j]	b	a	b	a	b	c	a
P[K+1]	P[0]	P[0]	P[1]	P[2]	P[3]	P[4]→ P[2]	P[0]
SP[j]=K	K=-1	K=0	K=1	K=2	K=3	K=1→-1	K=0

- L4: 일치되던 중 불일치가 발생하면 k값을 SP값으로 반복해 수정
- L5: 계속 일치 발생시 k값 증가
- ComputeSP는 이중 루프 구조로 $O(m^2)$이라고 판단하기 쉬움
  > 실제로는 L5에서 k를 1증가시키는 반면 L4에서 k 값을 감소시켜 O(m) 시간 내 처리 가능