String Matching Algorithms

Dae-Ki Kang

String Matching

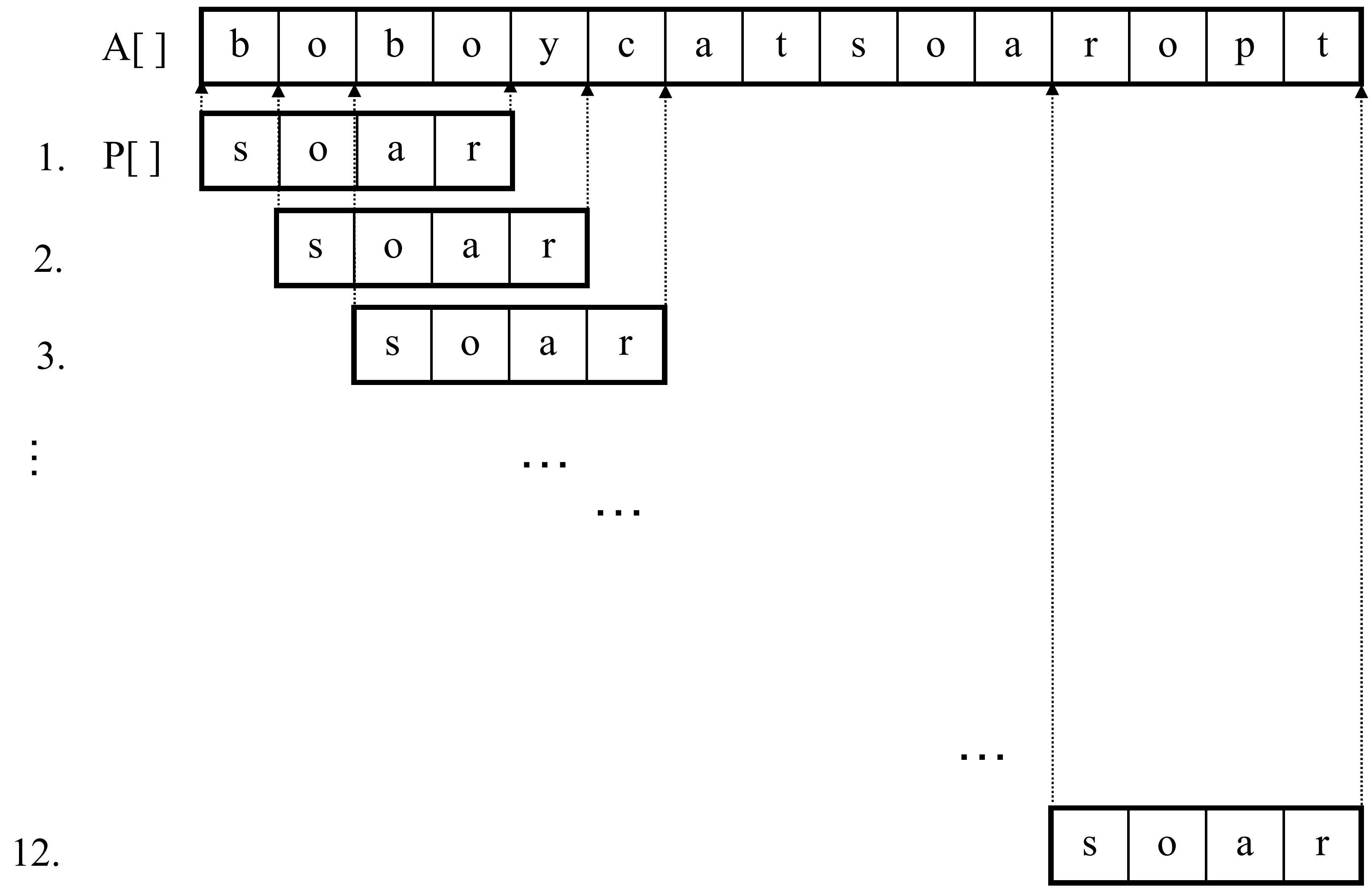
- Input
  - $A[1 \ldots n]$: Text string
  - $P[1 \ldots m]$: Pattern string
  - $m << n$
- Goal
  - Determine if $A[1 \ldots n]$ includes $P[1 \ldots m]$

Naïve Matching or Stupid Matching

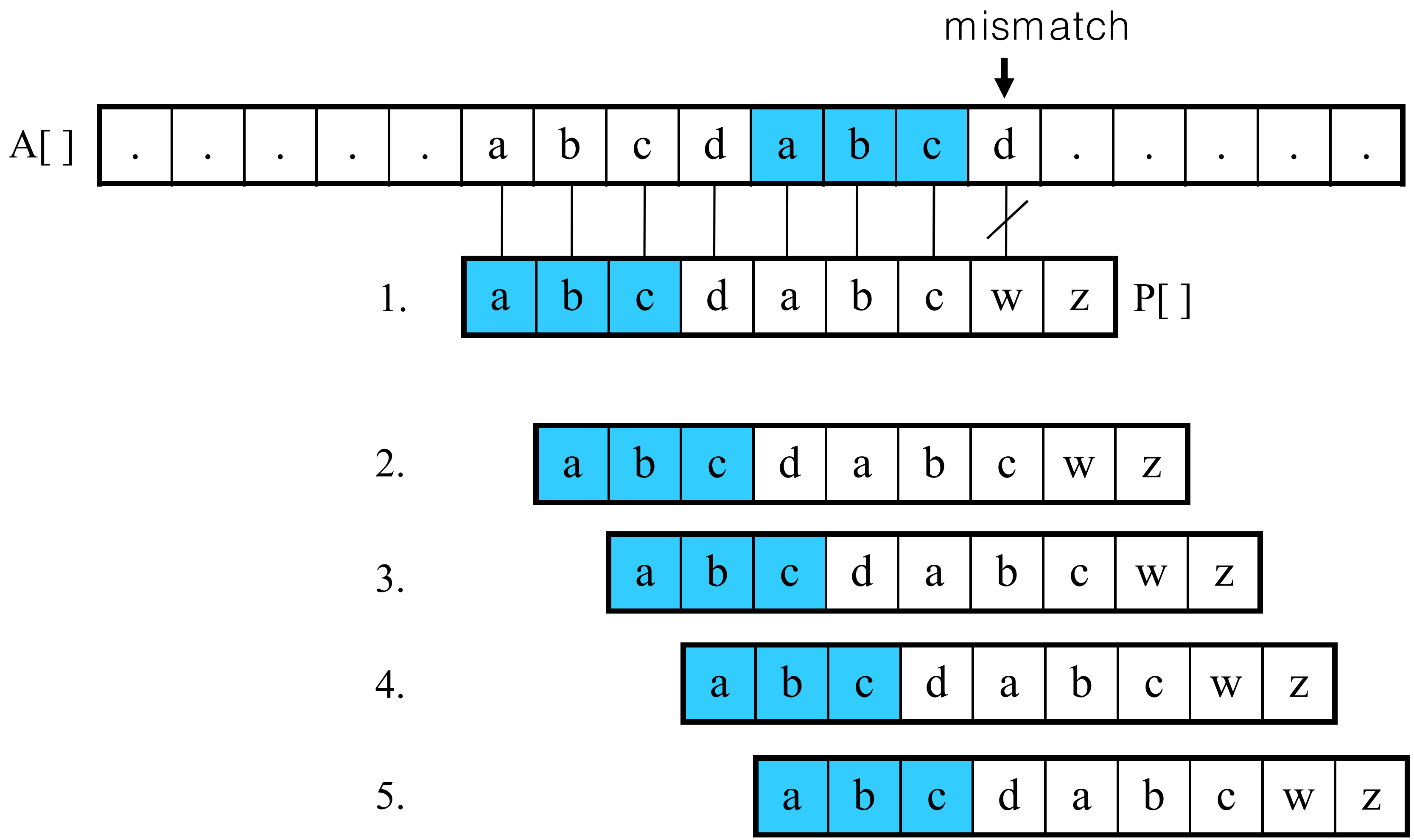
```
naiveMatching(A[ ], P[ ])
{
    ▷ n: length of A[ ], m: length of P[ ]
    for i ← 1 to n-m+1{
            if (P[1...m] = A[i...i+m-1]) then
                There is a match at A[i];
    }
}
```

✓ Running time: $O(mn)$

How Naïve Matching Works?



When Naïve Matching is inefficient?



Rabin-Karp Algorithm

- Change a pattern into a number and change a compared portion of a text string into a number
- Now, compare two numbers (instead of comparing two strings)
- String → Number
  - Depends on the cardinality of a character set $\sum$
  - Example: $\sum = \{a, b, c, d, e\}$
    - $|\sum| = 5$
    - Convert a, b, c, d, e into 0, 1, 2, 3, 4
    - String "cad" will be $2*5^2+0*5^1+3*5^0 = 28$

Overhead of the Conversion

- Conversion time of A[i⋯i+ m−1]
  - a_i = A[i+ m−1] + d(A[i+ m−2] + d(A[i+ m−3] + d(⋯ + d(A[i]))⋯)
  - $\Theta(m)$ time complexity
  - Total matching of A[1⋯n] takes $\Theta(mn)$
  - No better than naïve matching
- Horner's rule

  Regardless of m, calculate the number as follows:
  - $a_i = d(a_{i-1} - d^{m-1}A[i-1]) + A[i+m-1]$
  - We calculate d^{m-1} once
  - Two multiplications and two additions

One matching example

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

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

...

Basic Rabin Karp Algorithm

```
basicRabinKarp(A, P, d, q)
{
    ▷ n : length of A[ ],  m : length of P[ ]
    p ← 0; a1 ← 0;
    for i ← 1 to m {              ▷ calculate a1
          p ← dp + P[i];
          a1 ← da1 + A[i];
    }
    for i ← 1 to n−m+1{
          if (i ≠ 1) then ai ← d(ai−1 − d^(m−1)A[i−1]) + A[i+m−1];
          if (p = ai) then matched at A[i];
    }
}
```

✓ Total running time: $\Theta(n)$

Problem of basic Rabin Karp Algorithm

- Depending on $|\Sigma|$ and m, a_i can be big
  - bigger than CPU register size
  - and causes overflow
- Solution
  - Limit the size of a_i using modulo operation
  - $a_i = d(a_{i-1} - d^{m-1}A[i-1]) + A[i+m-1]$ →
    $b_i = (d(b_{i-1} - (d^{m-1} \bmod q)A[i-1]) + A[i+m-1]) \bmod q$
  - q is a big prime number, but dq should be fit in a register

Matching using modulo

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

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

…

Rabin-Karp Algorithm

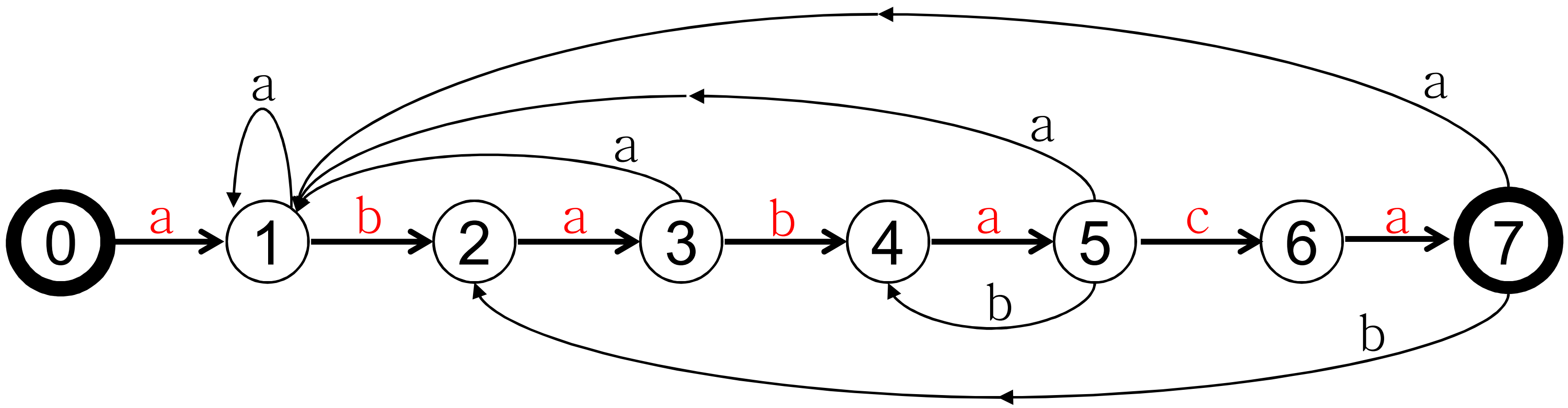
```
RabinKarp(A, P, d, q)
{
    ▷ n : size of A[ ],  m : size of P[ ]
     p ← 0; b1 ← 0;
    for i ← 1 to m {                                   ▷ b1
        p ← (dp + P[i]) mod q;
        b1 ← (db1 + A[i]) mod q;
    }
    h ← d^(m-1) mod q;
    for i ← 1 to n-m+1{
        if (i ≠ 1) then bi ← (d(bi-1 − hA[i-1]) + A[i+ m-1]) mod
    q;
        if (p = bi) then
                if (P[1…m] = A[i…i+ m-1]) then
                        there is a match at A[i];
    }
}
```

✓ Average running time: $\Theta(n)$

Automata Based Matching

- Finite Automata
  - Finite symbols, states and transition
  - Five tuple: $(Q, q0, A, \sum, \delta)$
    - Q : states
    - Q0 : start state
    - A : accepted states
    - $\sum$: input alphabet
    - δ : state transition diagram
- States represent a snapshot of matching process

Automata that checks ababaca



S: dvganbbactababaababacabababacaagbk…

Implementation of Automata

State \ Input symbol	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

➡

State \ Input symbol	a	b	c	Else
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

Algorithm that checks matching using automata

```
FA-Matcher (A, δ, f)
▷ f : accepted state
{
    ▷ n: length of A[ ]
    q ← 0;
    for i ← 1 to n {
        q ← δ(q, A[i]);
        if (q = f) then we found a match at A[i−m+1];
    }
}
```

✓ Total running time: $\Theta(n + |\Sigma|m)$

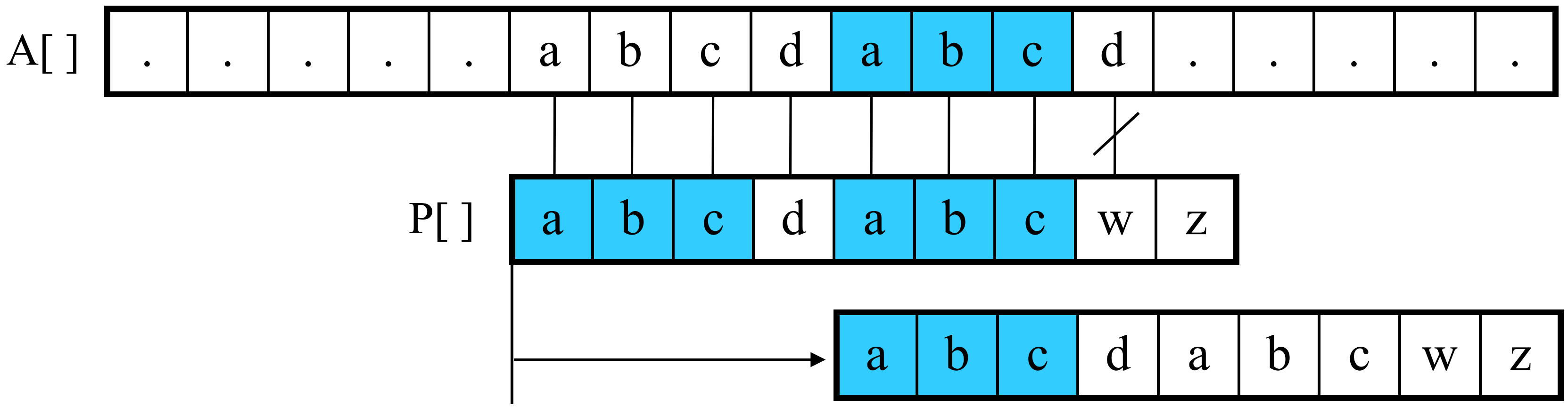
Compute Transition Function

```
Compute_Transition_Function(P, ∑)
{
    m ← |P|; ▷ length of pattern
    for q ← 0 to m do
        for each character a ∈ ∑ do
                k ← min(m+1,q+2);
                repeat k ← k - 1
                until P[1..k] is a suffix of P[1..q]a;
                δ(q, a) ← k;
}
```

✓Running time: $\Theta(|\sum|m)$

KMP(Knuth-Morris-Pratt) Algorithm

- Similar to automata based matching
- Commonalities
  - Prepare states for mismatch
  - Simpler than automata based match



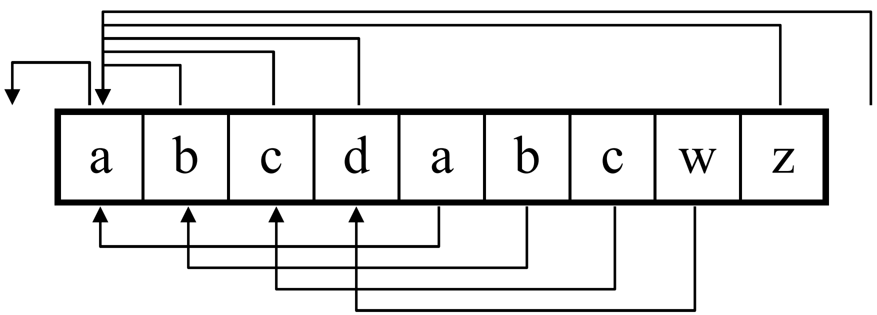
Prepare return points for each mismatch

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

$\pi[8] = 4$

Matched upto “abcdabc”, failed at “w”
“abc” at 1,2,3 and “abc” at 5,6,7 are the same.
So, we compare mismatched text char with P[4]

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



For each index of pattern,
we prepare the return point.

KMP Algorithm

```
KMP(A[ ], P[ ])
{
    preprocessing(P);
    i ← 1;  ▷ index pointer of text
    j ← 1;  ▷ index pointer of pattern
    ▷ n: size of A[ ], m: size of P[ ]
    while (i ≤ n) {
        if (j = 0 or A[i] = P[j])
                then { i++ ;  j++ ; }
                else  j ← π [j];
        if (j = m+ 1) then {
                There is a match at A[i−m];
                j ← π [j];
        }
    }
}
```

✓Running time: $\Theta(n)$

Preprocessing

```
preprocessing(P)
{
    m ← |P| ; ▷ length of pattern
    π [1] ← 0;
    k ← 0;
    for q ← 2 to m do
            while (k > 0) and (P[k+1] ≠ P[q]) do
                        k ← π [k];
            if (P[k+1] = P[q]) then k ← k+1;
            π [q] ← k;
    return π;
}
```

✓Running time: $\Theta(m)$