기본 알고리즘 제8장

2017-Fall

국민대학교 컴퓨터공학부 최준수

String Matching

- ## Substring search
  - Find pattern of length M in a text of length N. (typically $N \gg M$)
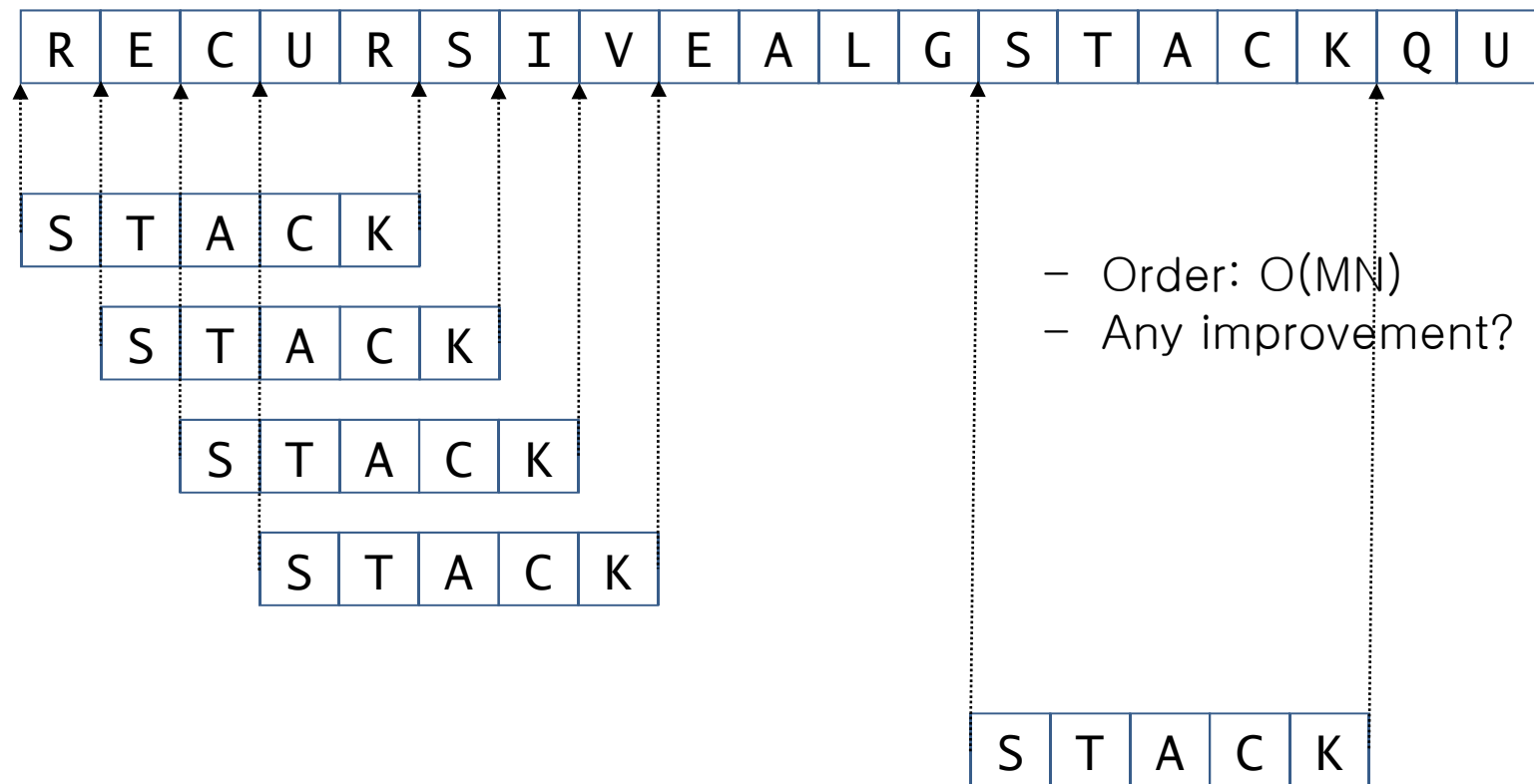
text → R E C U R S I V E A L G S T A C K Q U

←—————→
match

pattern → S T A C K

Brute-Force Substring Search

- ## Naïve Algorithm 1
  - Check for pattern starting at each text position

R	E	C	U	R	S	I	V	E	A	L	G	S	T	A	C	K	Q	U

S	T	A	C	K

S	T	A	C	K

S	T	A	C	K

S	T	A	C	K

- Order: O(MN)
- Any improvement?

S	T	A	C	K

3

Brute-Force Substring Search

- ## Naïve Algorithm 2
  - Check for pattern starting at each text position

```c
int naiveStringMatch(char text[], char pattern[])
{
    int patLength, txtLength;

    patLength = strlen(pattern);
    txtLength = strlen(text);

    for(int i=0; i <= txtLength – patLength; i++)
    {
        for(int j=0; j < patLength; j++)
            if(text[i+j] != pattern[j])
                break;
        if(j == patLength)
            return i;
    }
    return 0;
}
```

Find just only one sample of pattern.
Modify the code to find all samples of the pattern.

Brute-Force Substring Search

- ## Naïve Algorithm 2
  - Check for pattern starting at each text position

Brute-Force Substring Search

- ## Naïve Algorithm 2
  - Naïve algorithm can be slow if text and pattern are repetitive

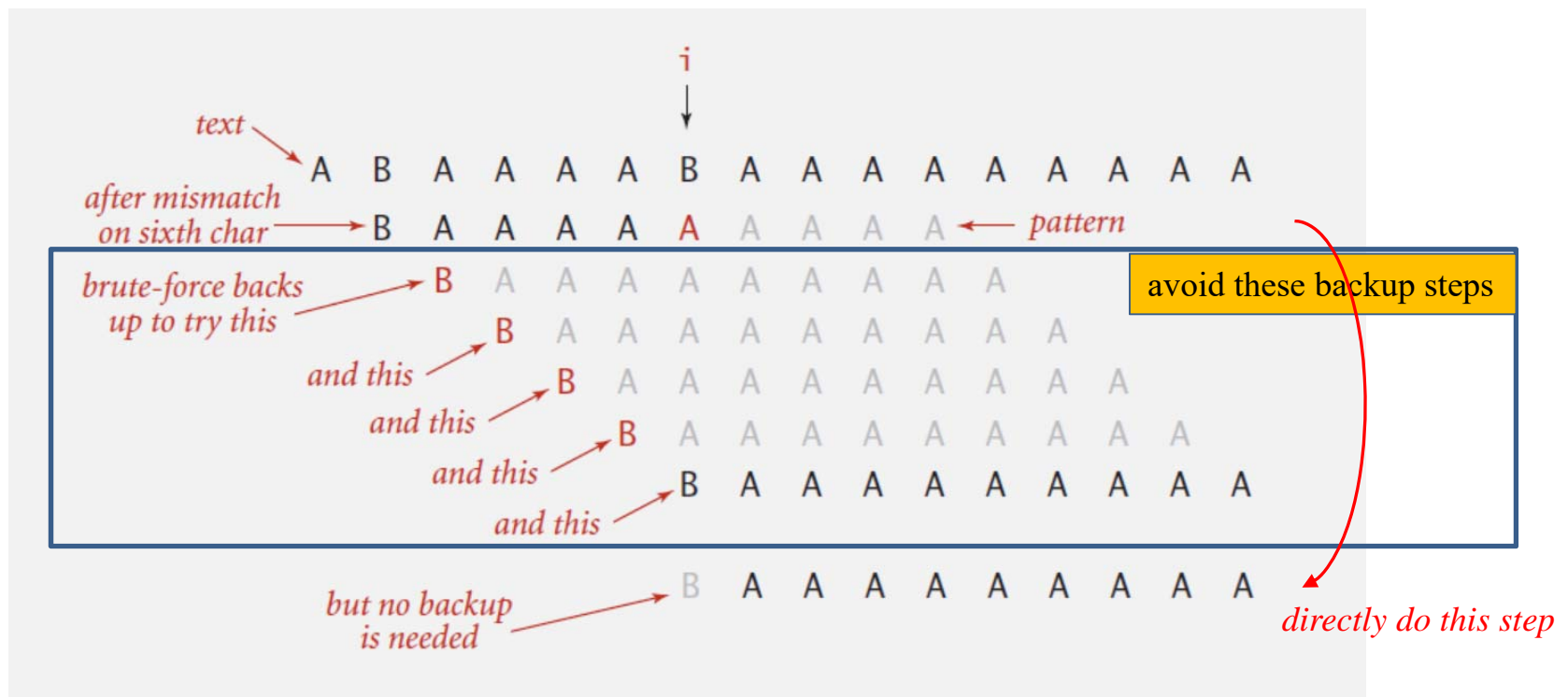- Still order: O(MN)
- Any improvement?
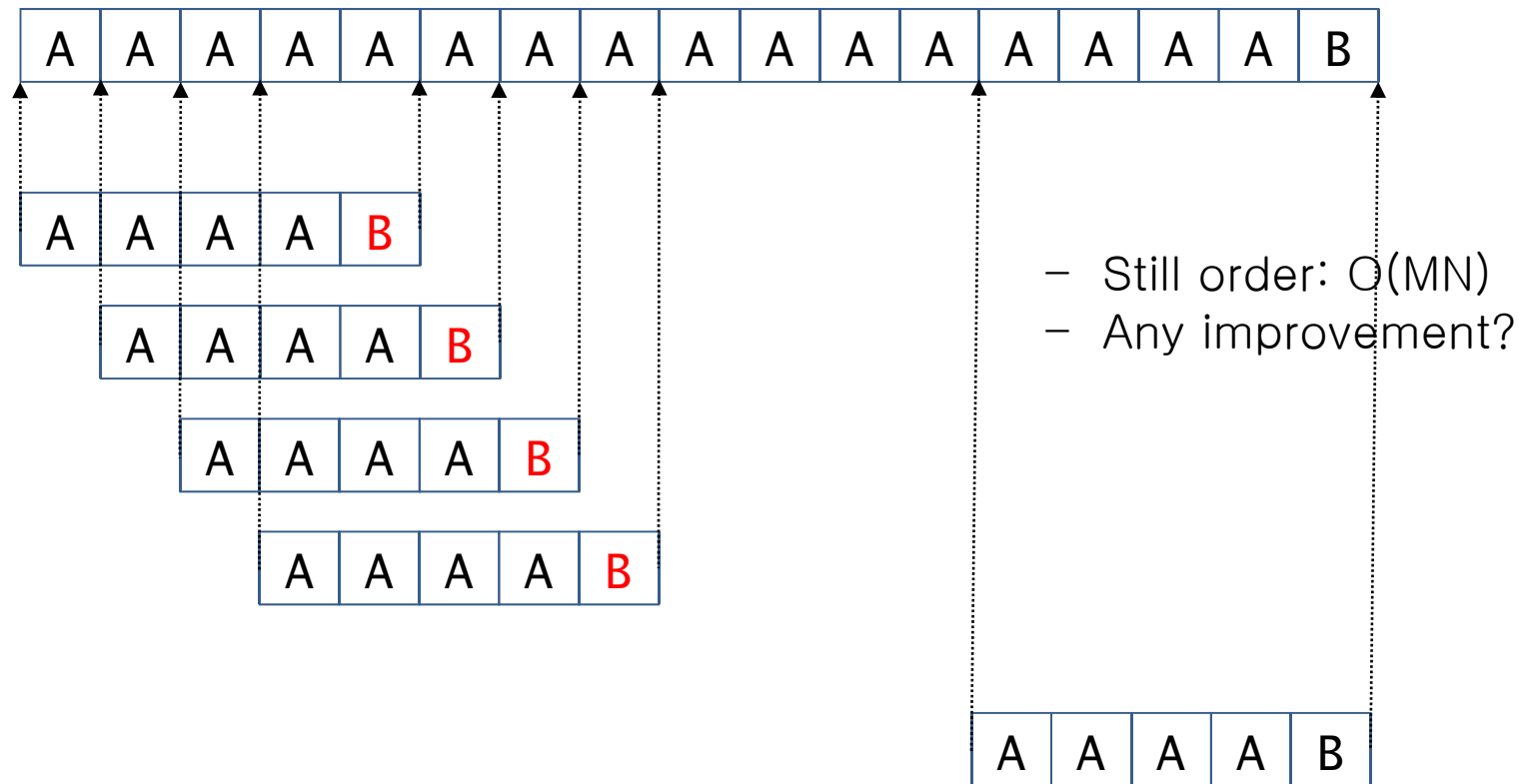
Brute-Force Substring Search

- ## Improvement
  - ### Develop a linear time algorithm
  - ### Avoid backup
    - Naïve algorithm needs backup for every mismatch
    - Thus naïve algorithm cannot be used when input text is a stream.
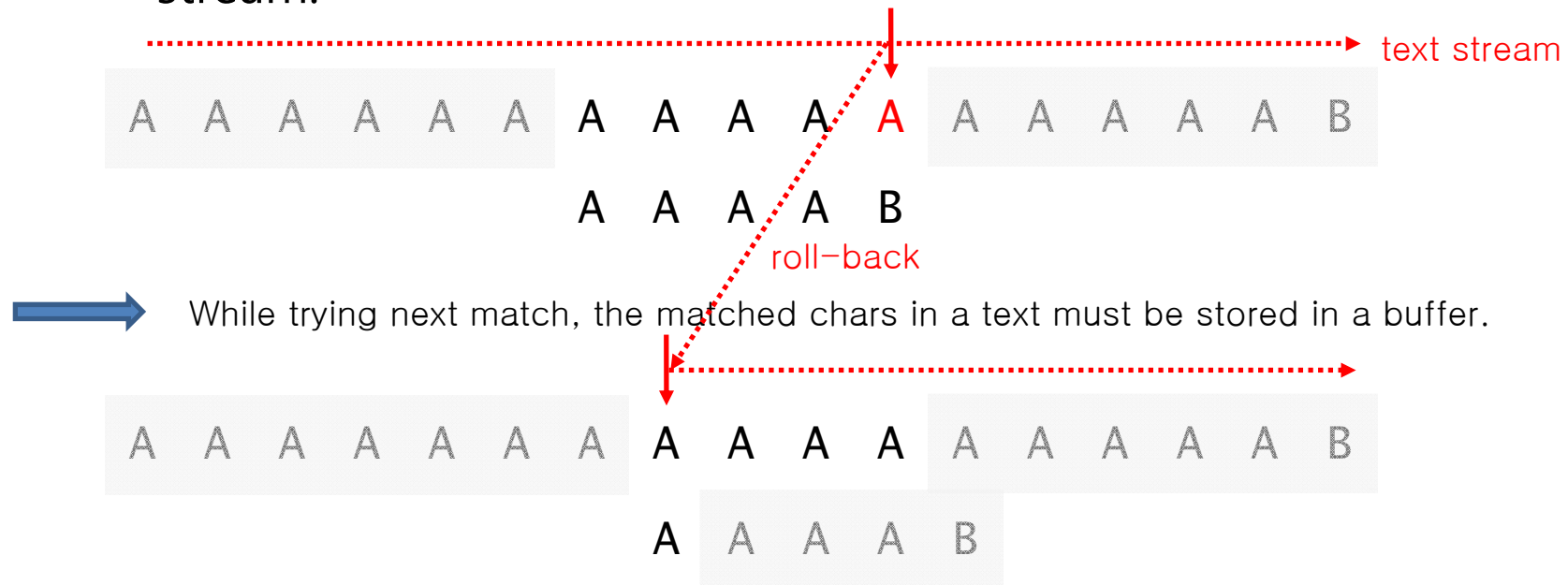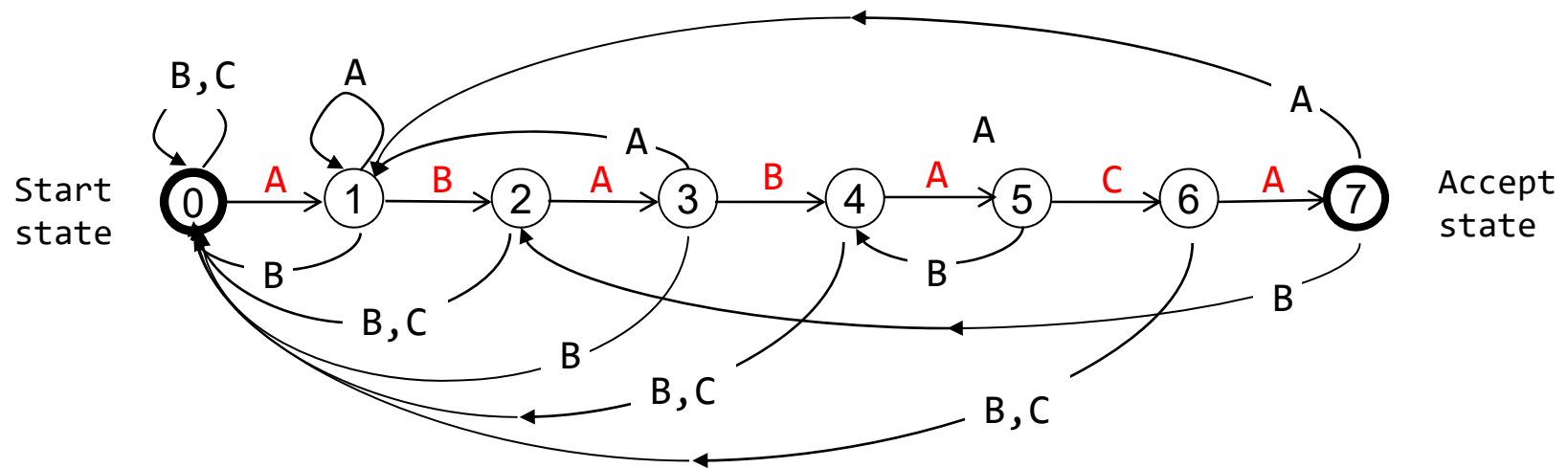
Knuth-Morris-Pratt(KMP) Algorithm

- ## KMP algorithm
  - Clever method to always avoid backup problem.
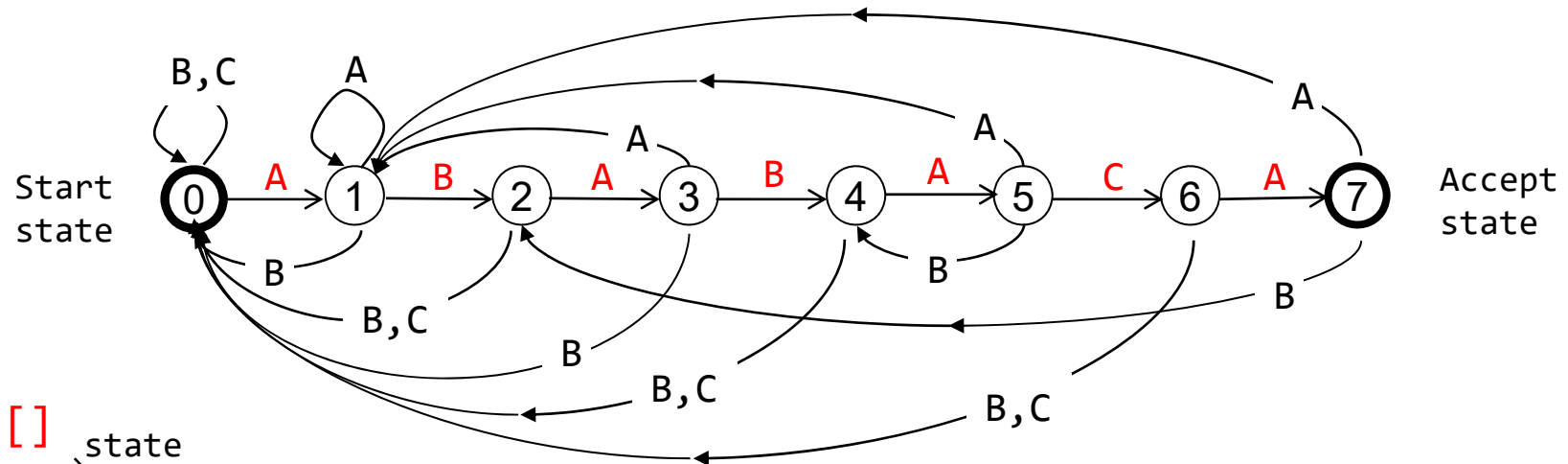
Deterministic Finite Automaton

- ## DFA(Deterministic Finite State Automaton)
  - Finite number of states (including start and accept states)
  - Exactly one transition for each char
  - Accept if sequence of transitions leads to accept state

DFA for pattern ABABACA

DFA

DFA for pattern ABABACA

DFA[][]

char \ state	0	1	2	3	4	5	6	7
A	1	1	3	1	5	1	7	1
B	0	2	0	4	0	4	0	2
C	0	0	0	0	0	6	0	0
others	0	0	0	0	0	0	0	0

```
if in state j reading char c:
    if j is 7, halt and accept
    else move to state DFA[c][j]
```

Example:
    text: ABCABAABABABACACA
    state:01201231234 54567

DFA
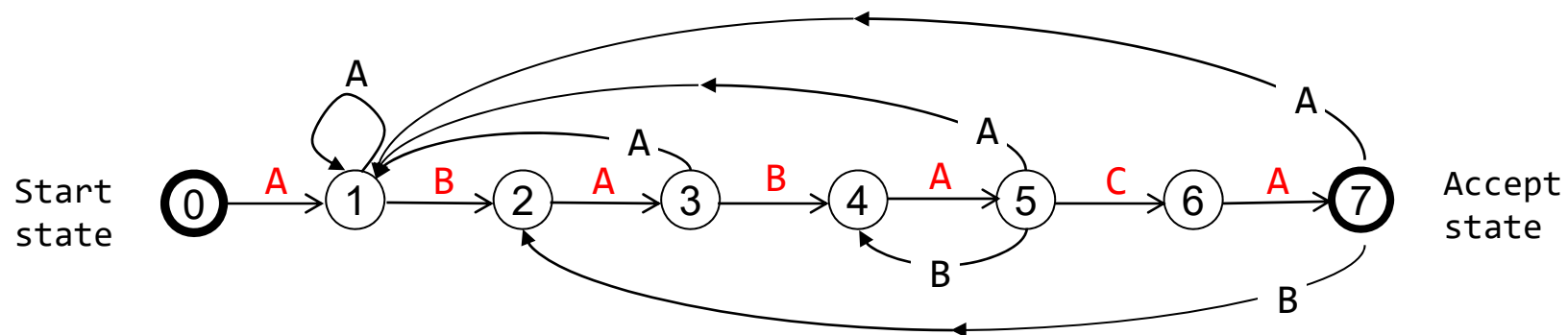
DFA for pattern ABABACA
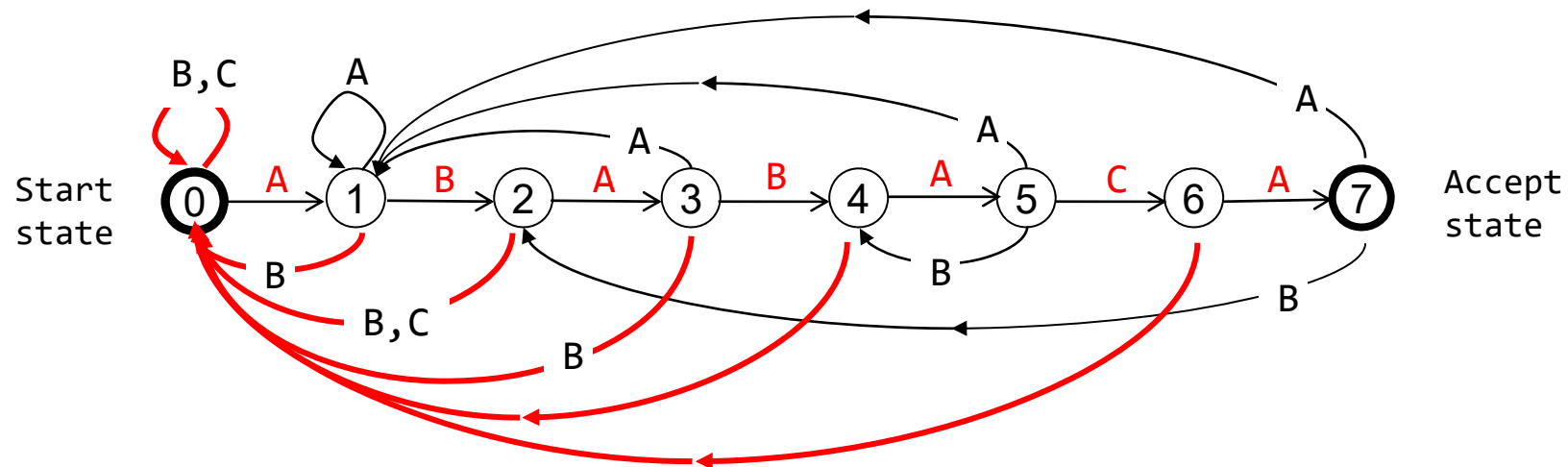
Simplified Diagram: remove transitions to state 0

Algorithm with DFA

- Difference from naïve algorithm
  - Precomputation of DFA[][] from pattern
  - Text pointer `i` never decrements (no backup)

```
// patLength = strlen(pattern);
int KMP(char text[])
{
    int i, j, txtLength;

    txtLength = strlen(text);

    for(i=0, j=0; i <= txtLength && j < patLength; i++)
        j = DFA[text[i]][j];  // text[i] to be modified

    if(j == patLength)
        return i - patLength;
    else
        return 0;
}
```
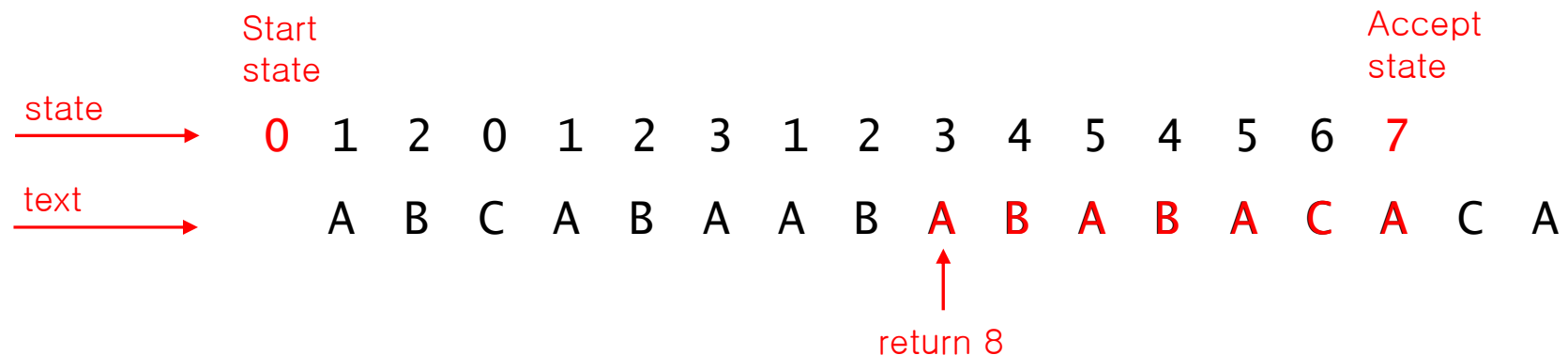 - Order: O(N)
    simulation of DFA on text with no backup
  - How to build DFA efficiently?

12

DFA for pattern ABABACA
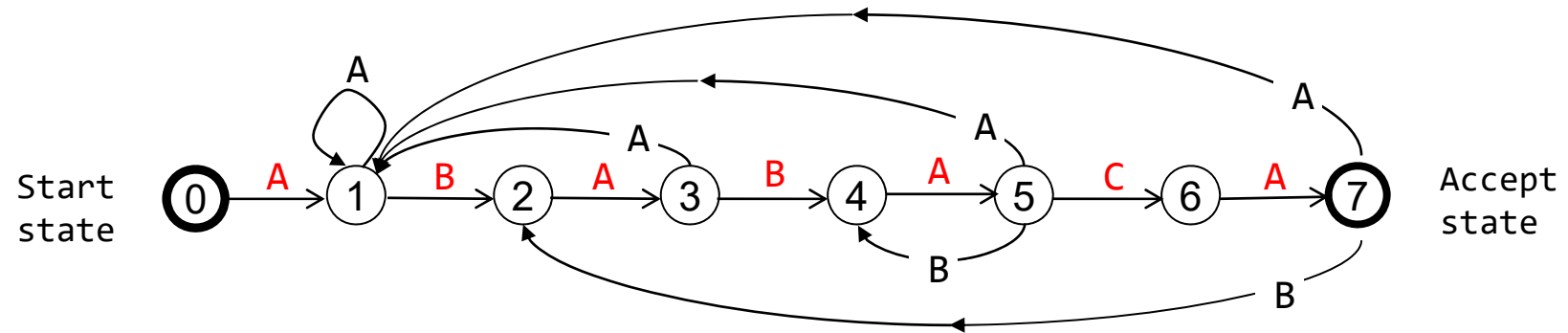
Interpretation of DFA

- The state of DFA represents
  - the number of characters in pattern that have been matched

State 5

text → B C B A A B A B A A B A

Suffix of txt[8]

pattern → A B A B A C A

Prefix of pat[]
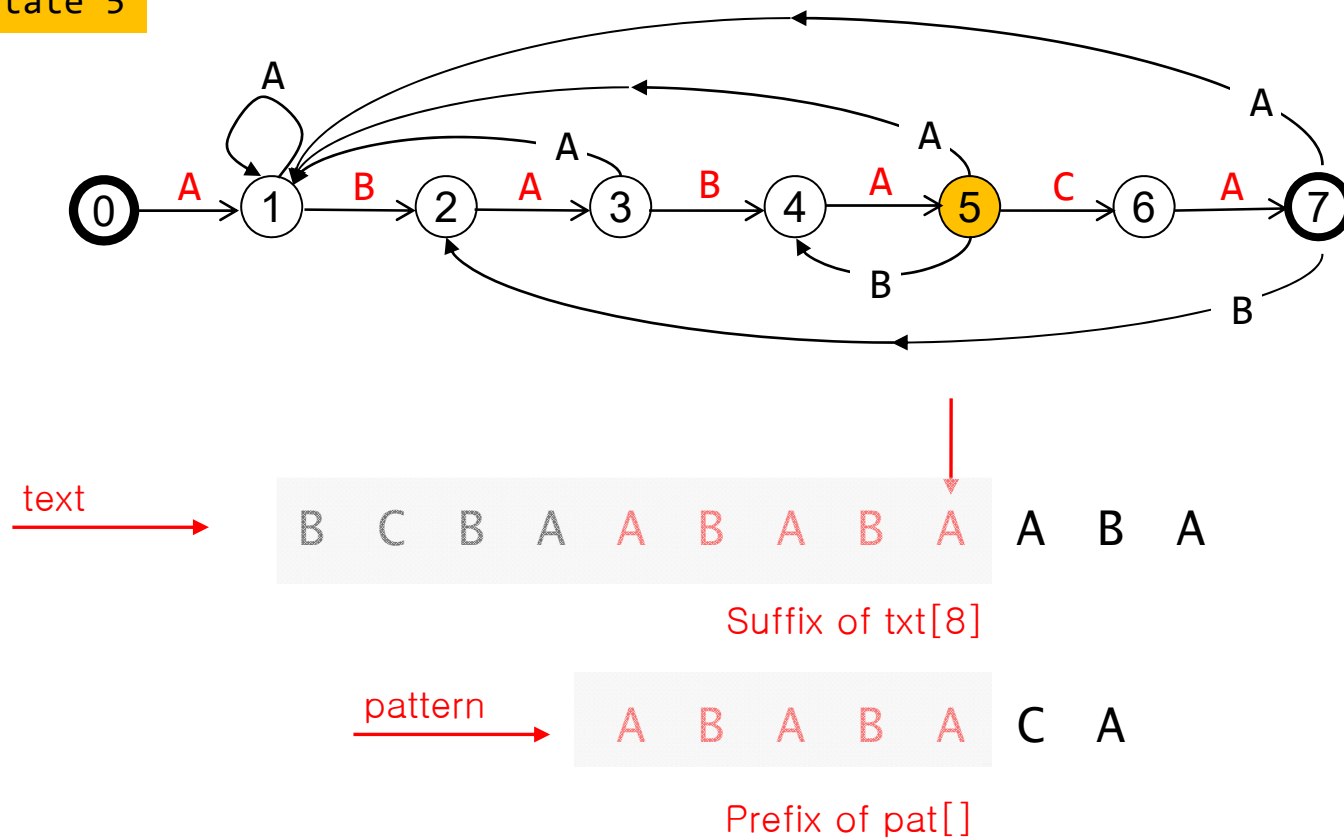
DFA Construction

- ## DFA Construction:
  - Suppose that all transitions from state `0` to state `j-1` are already computed
  - Match transition:
    - If in state `j` and next char `c  == pat[j]`, then transit to state `j+1`.

Pattern: **ABABACA**

State 5
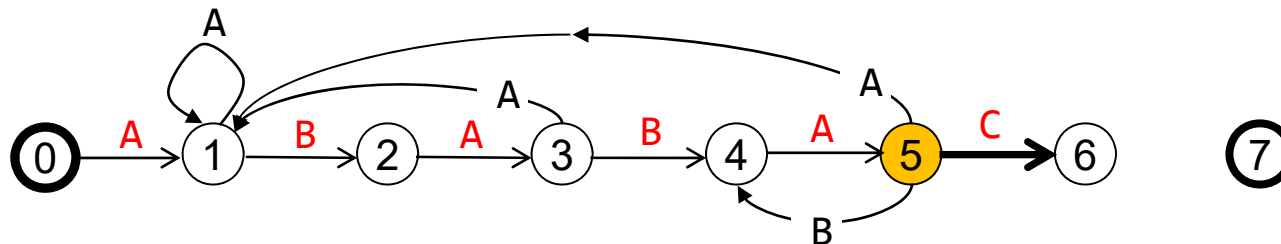
DFA Construction

- ## DFA Construction:
  - ### Mismatch transition:
    - If in state `j` and next char `c != pat[j]`, then which state to transit?

Pattern: **ABABA****C****A**

State 5

DFA Construction

Matching start here   mismatch

text → B C B A A B A B A A B A

pattern → A B A B A C A

roll-back

Roll back: Next matching start here from state 0

text → B C B A A B A B A A B A
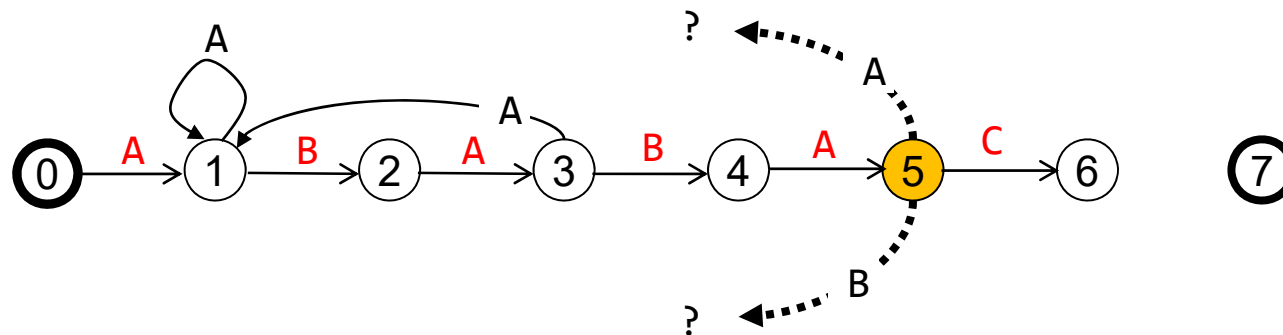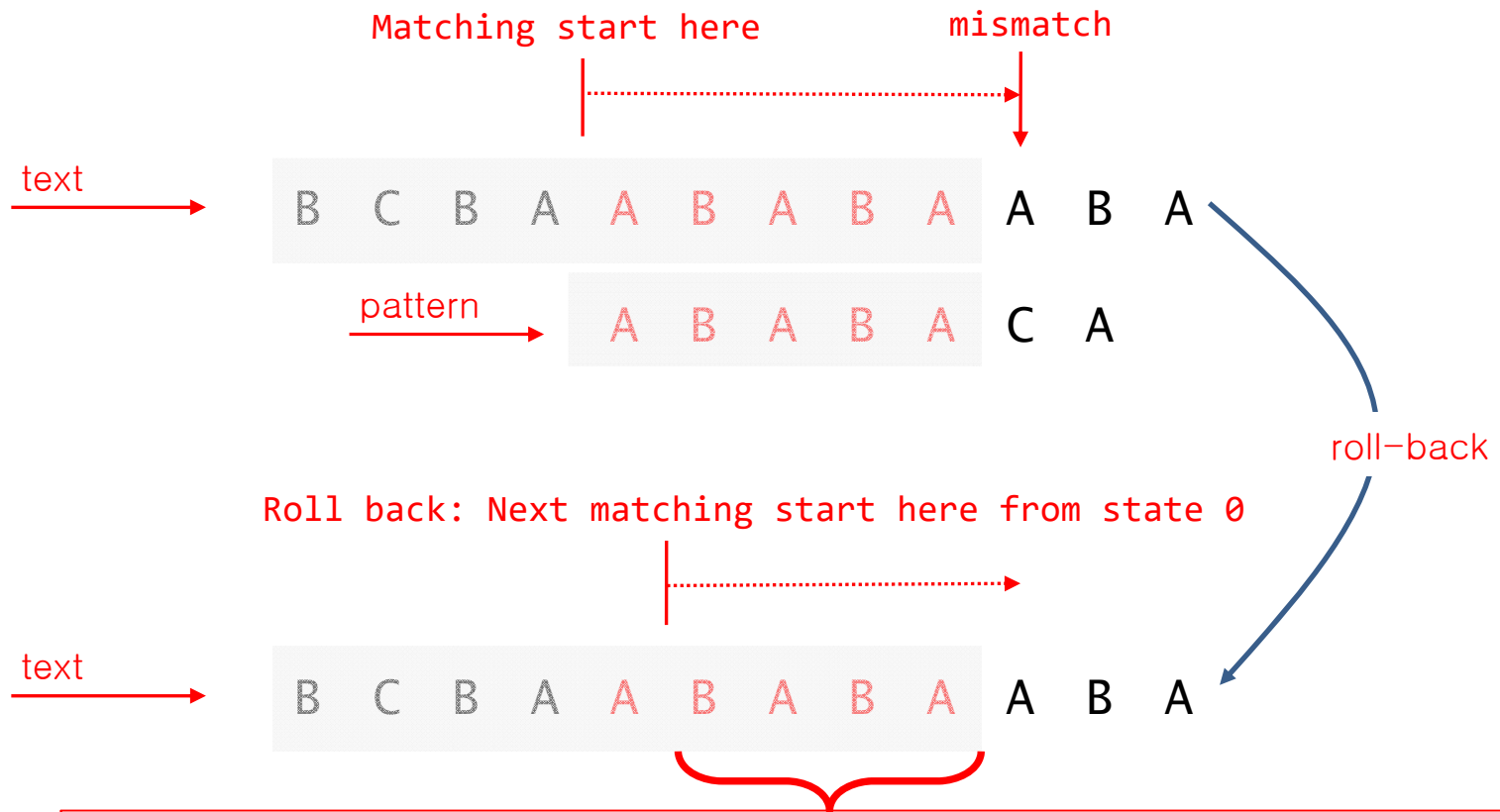
- *The same as pat[1] ~ pat[j-1]*
- Roll-back and transit to some state X by matching
    pat[1] ~ pat[j-1] from state 0 on DFA.
- Transit to the next state DFA['A'][X] for the mismatched char 'A'.

DFA Construction

- ## DFA Construction:
  - ### Mismatch transition:
    - If in state `j` and next char `c != pat[j]`, then the last `j-1` characters of input text are `pat[1] ~ pat[j-1]`, followed by `c`.
  - ### Compute `DFA[c][j]`:
    - Simulate `pat[1] ~ pat[j-1]` on DFA from state 0 and let X be the current state
    - Then `DFA[c][j] = DFA[c][X]`

DFA Construction

- DFA Construction:
  - Mismatch transition:
    - DFA[c][j] = DFA[c][X]

$$DFA['A'][5] = DFA['A'][3] = 1$$
$$DFA['B'][5] = DFA['B'][3] = 4$$

Pattern: ABABACA

State 5

X=3
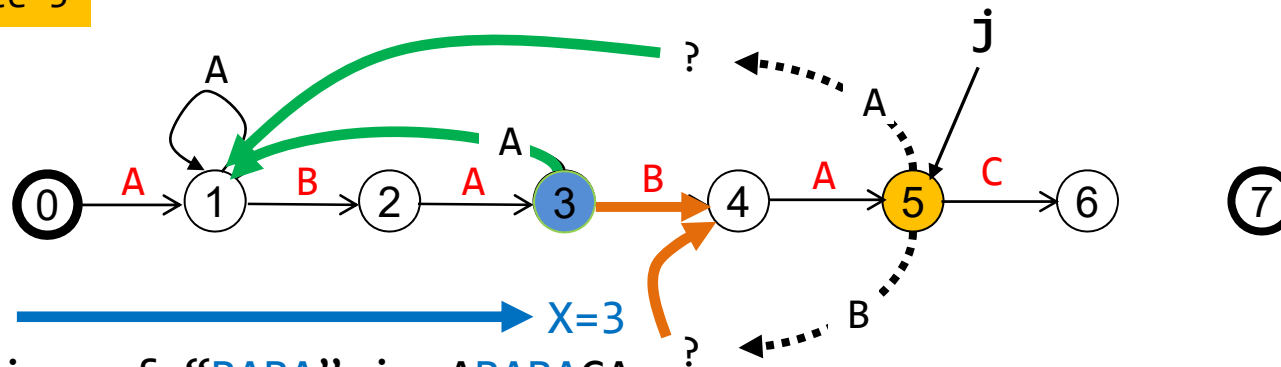
Simulation of "BABA" in ABABACA

DFA Construction

- ## DFA Construction:
  - ### Mismatch transition:
    - If in state `j` and next char `c != pat[j]`, then the last `j-1` characters of input text are `pat[1] ~ pat[j-1]`, followed by `c`.
  - ### To compute `DFA[c][j]`:
    - Simulate `pat[1] ~ pat[j-1]` on DFA (*still under construction*) and let the current state `X`.
    - take a transition `c` from state `X`.
    - Running time : require `j` steps.
    - But, if we maintain state `X`, it takes only constant time!

DFA Construction

- ## DFA Construction:
  - ## Maintaining state X:
    - Finished computing transitions from state `j`.
    - Now, now move to next state `j+1`.
    - Then what the new state(`X'`) of `X` be?

State 6

Pattern: ABABACA

j+1

X=3

Simulation of "BABA" in ABABACA

X' = ?
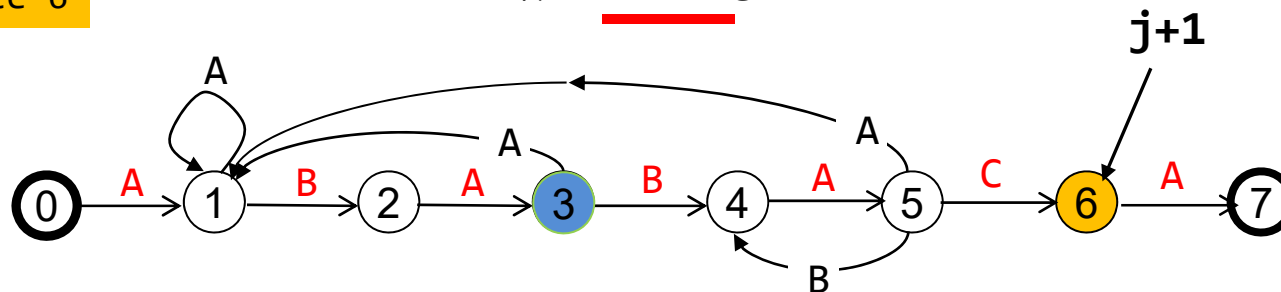
Simulation of "BABAC" in ABABACA

- Simulation requires j+1 steps
- But, `X' = DFA['C'][X]`

`X' = DFA['C'][3] = 0`

20

DFA Construction

- DFA Construction: A Linear Time Algorithm
  - For each state `j`:
    - Match case: set `DFA[pat[j]][j]=j+1`
    - Mismatch case: Copy `DFA[][X]` to `DFA[][j]`
    - Update `X`

```c
int DFA[MAX_SIZE][MAX_SIZE];  /* initially all elements are 0 */
// int R;          /* text character set size */

void constructDFA(char pat[])
{

    int patLength = strlen(pat);

    DFA[pat[0]][0] = 1;

    for(int X=0, j=0; j<patLength; j++)
    {
        for(int c=0; c<R; c++)                  // copy mismatch cases
            DFA[c][j] = dfa[c][X];

        DFA[pat[j]][j] = j+1;                   // copy match case
        X = DFA[pat[j]][X];                     // update X
    }
}
```

DFA Construction

- DFA Construction: Example

DFA[][]

j start from 1

0123456
Pattern: ABABACA

char \ state	0	1	2	3	4	5	6	7
A	1	1	3	1	5	1	7	1
B	0	2	0	4	0	4	0	2
C	0	0	0	0	0	6	0	0
others	0	0	0	0	0	0	0	0

X   0   0   1   2   3   0   1   1

DFA[B][0]   DFA[B][1]   DFA[C][3]   DFA[A][1]

DFA[A][0]   DFA[A][2]   DFA[A][0]

Algorithm with DFA

- Question:
  - Text에 나타나는 모든 pattern 을 찾을 수 있는가?
    - Text: AAAAAAAAA
    - Pattern: AAAAA
    - 해답: 0, 1, 2, 3, 4, 5

Prefix/Suffix

- Prefix / Suffix of a Text

bananada	

Prefix	Suffix
bananada	bananada
bananada	bananada
bananada	bananada
bananada	bananada
bananada	bananada
bananada	bananada
bananada	bananada
bananada	bananada
bananada	bananada

NULL string ⟶

KMP Algorithm

- Naïve algorithm again:

First mismatch

i=5

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

text   A B A B A B A B A C A B

pattern A B A B A C A

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |

j=5
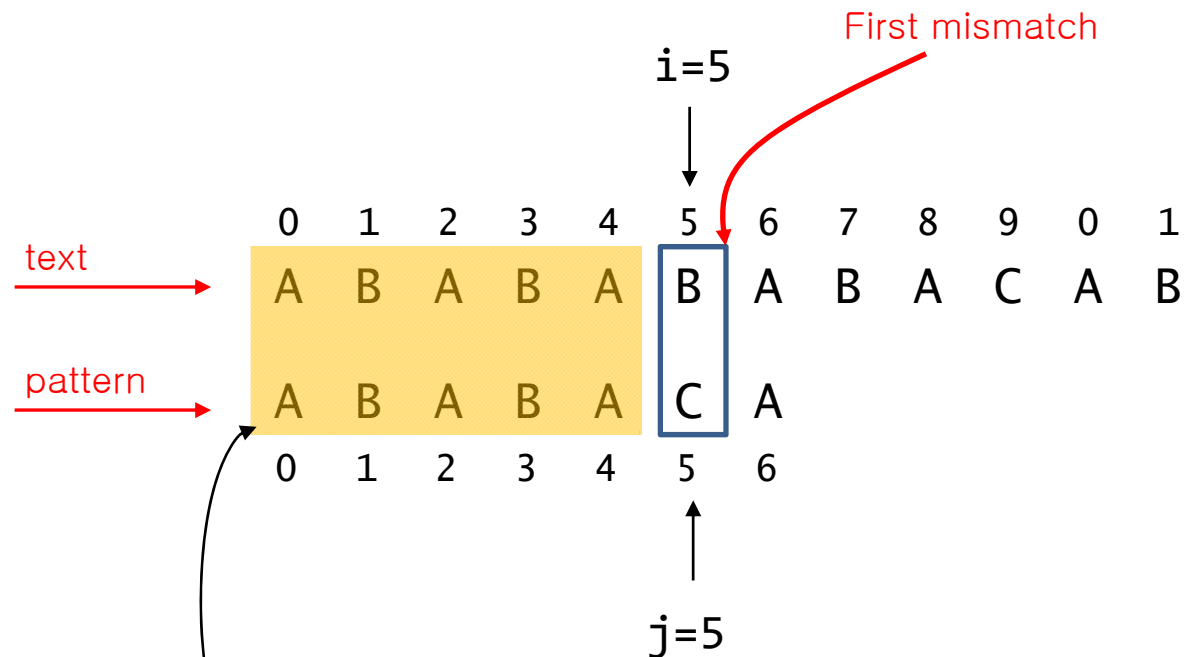
These matched characters have very important information.
Do not waste this information!!!

KMP Algorithm

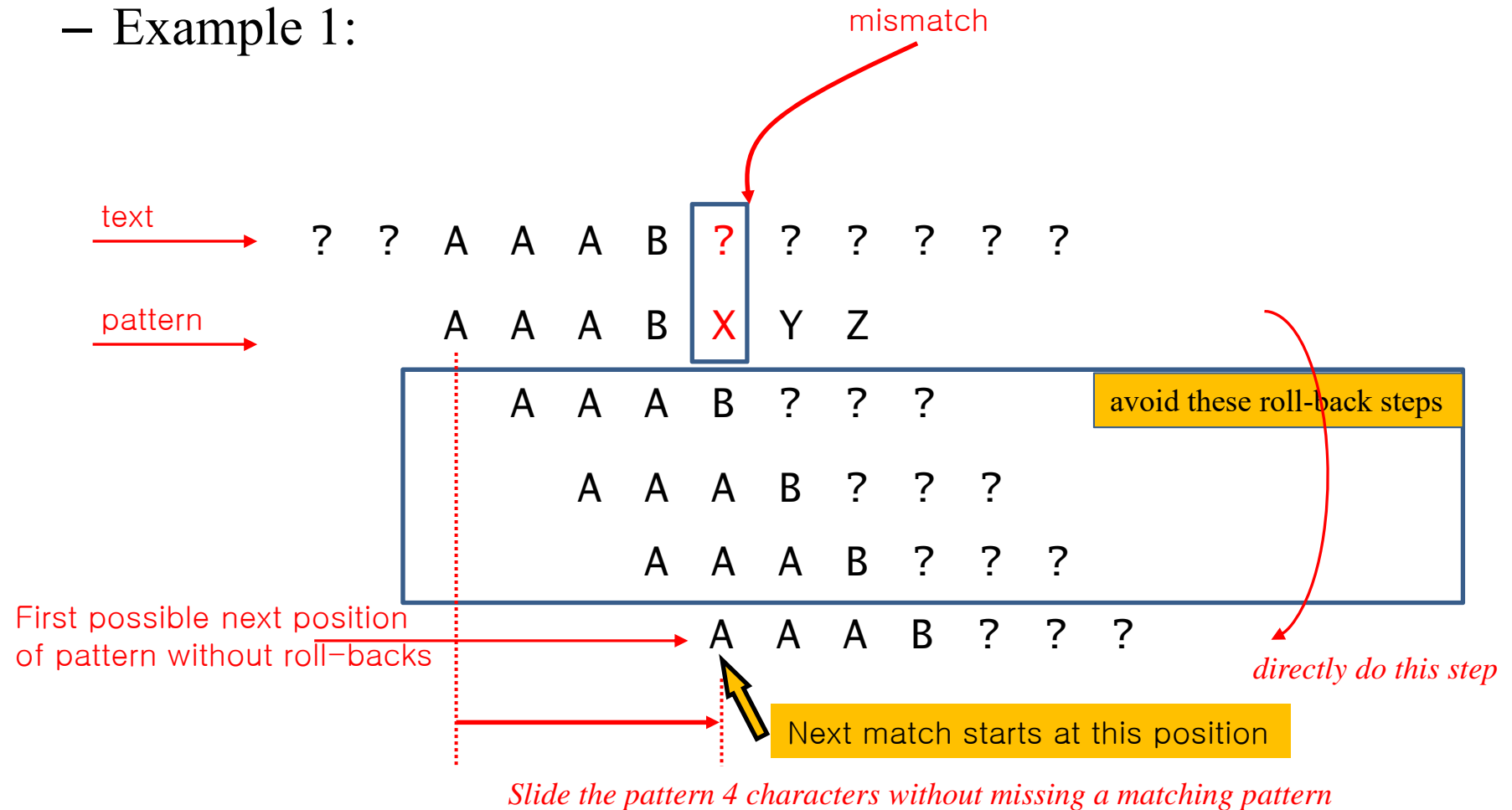- Avoid roll-backs in naïve algorithm:
  - Example 1:

mismatch

text → ? ? A A A B **?** ? ? ? ? ?

pattern → A A A B **X** Y Z

A A A B ? ? ?　　avoid these roll-back steps

A A A B ? ? ?

A A A B ? ? ?

First possible next position
of pattern without roll-backs → A A A B ? ? ?

directly do this step

Next match starts at this position

Slide the pattern 4 characters without missing a matching pattern

KMP Algorithm

- Avoid roll-backs in naïve algorithm:
  - Example 2:
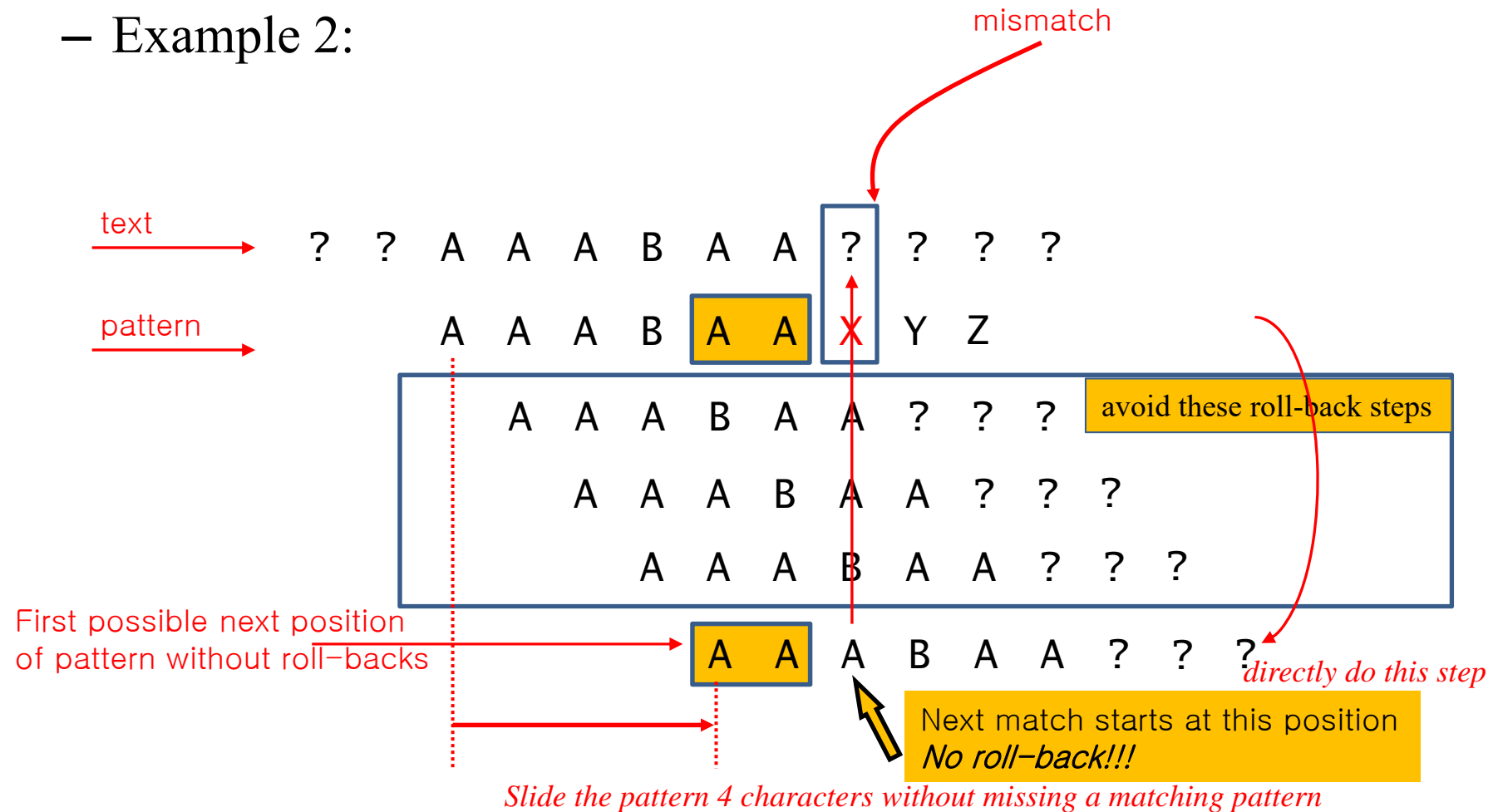
KMP Algorithm

- Prefix and Proper Suffix of the Prefix

pattern →

A A A B A A X Y Z

Prefix
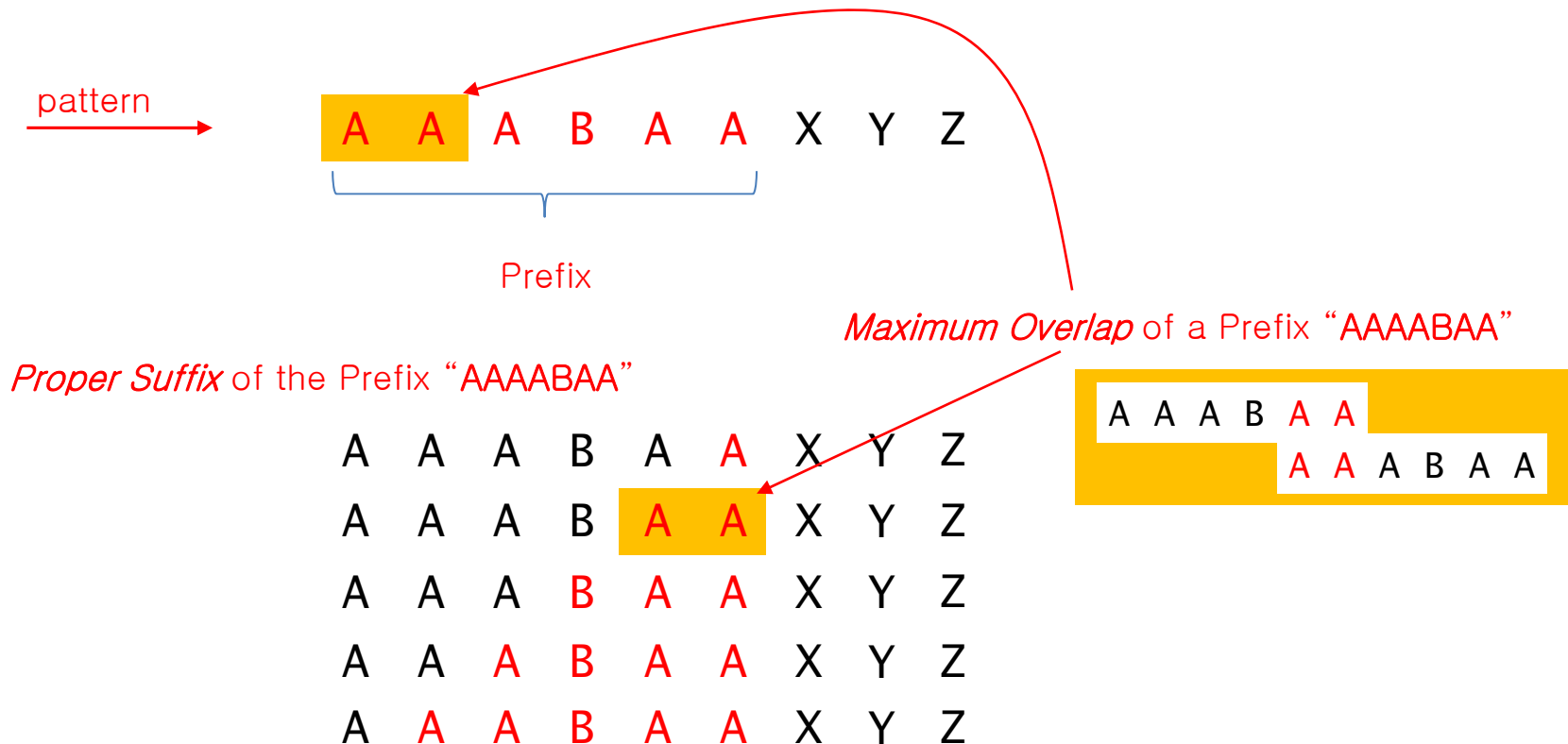
Proper Suffix of the Prefix "AAAABAA"

A A A B A A X Y Z

A A A B A A X Y Z

A A A B A A X Y Z

A A A B A A X Y Z

A A A B A A X Y Z

A A A B A A X Y Z ←

Not a *proper* suffix of the prefix
(the same as the prefix)

KMP Algorithm

- ## Maximum Overlap of a Prefix
  - the longest proper suffix that is equal to prefix of the prefix

pattern → A A A B A A X Y Z

Prefix

Proper Suffix of the Prefix "AAAABAA"

Maximum Overlap of a Prefix "AAAABAA"

A A A B A A X Y Z
A A A B A A X Y Z
A A A B A A X Y Z
A A A B A A X Y Z
A A A B A A X Y Z

A A A B A A
    A A A B A A

KMP Algorithm

- ## Maximum Overlap of a Prefix
  - the longest proper suffix that is equal to prefix of the prefix

- ## Example:

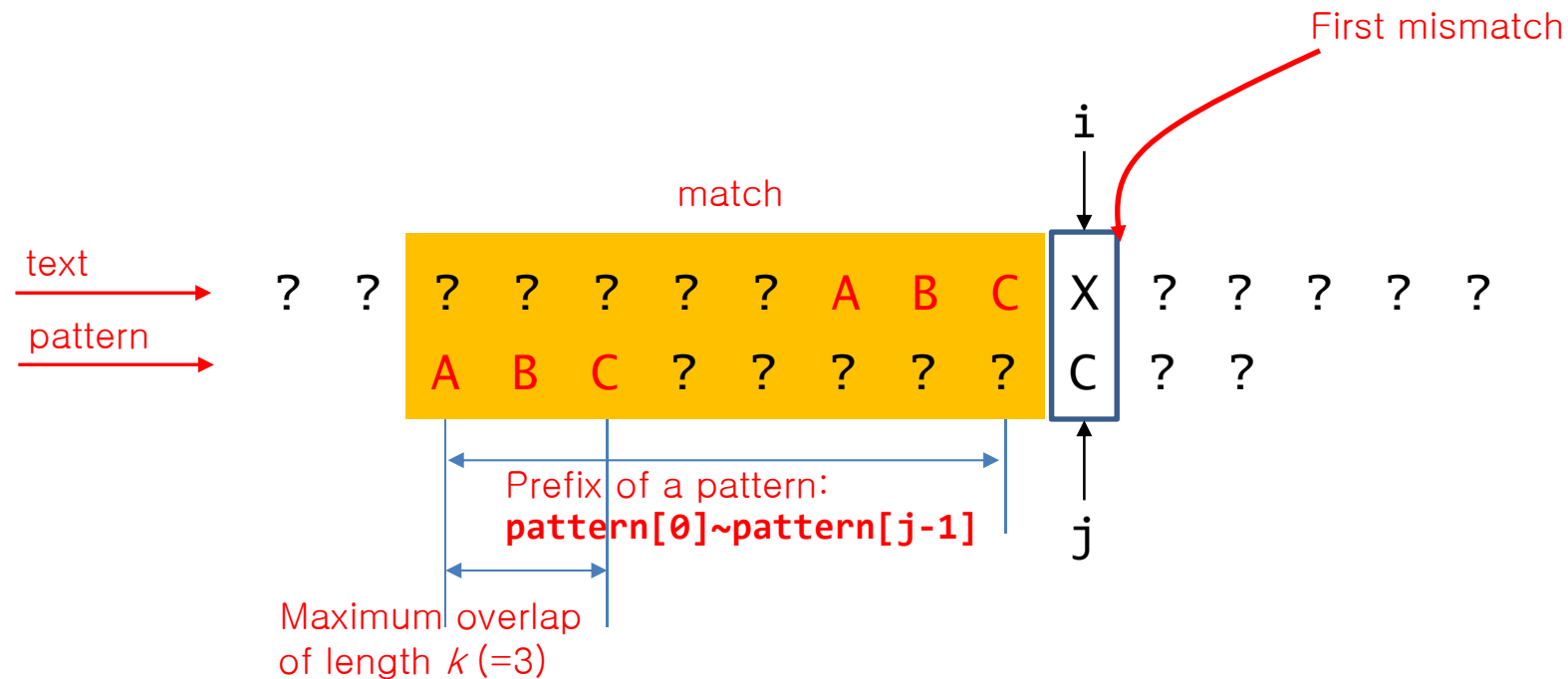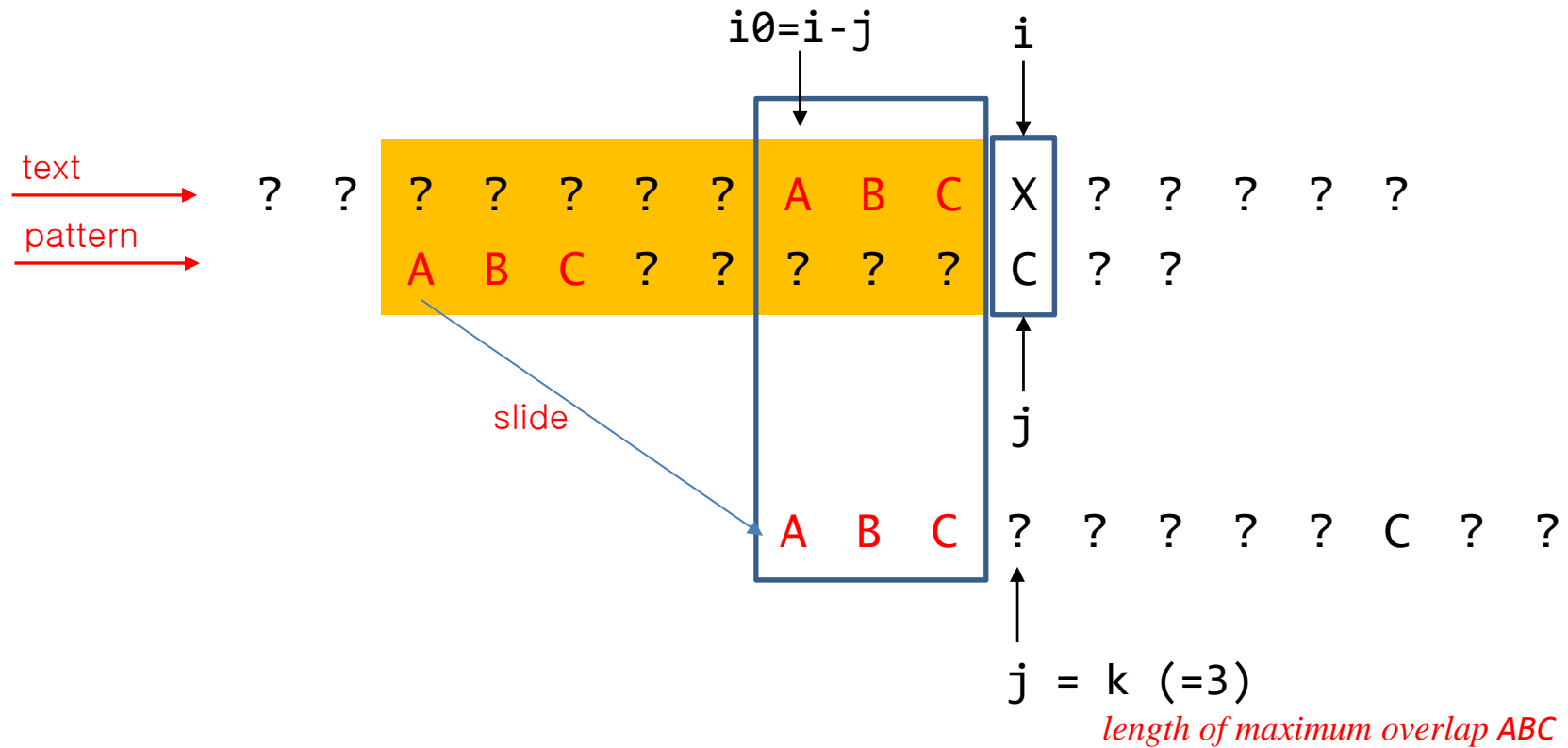| Prefix	Maximum Overlap
AAAAA	AAAA
AABA	A
AAAB	
ABABABAB	ABABAB

not **AAAAA**

NULL String

- Reuse of prefix information when there is a mismatch
  - Mismatch at `text[i]` and `pattern[j]`

Note that if the mismatched *location* is `pattern[j]`, then *prefix* is:
`pattern[0]~pattern[j-1]`

KMP Algorithm

– Then we can slide the pattern so that the *suffix and prefix aligns without missing out on a match*:

$i0=i-j$

i

text

pattern

? ? ? ? ? ? ? A B C X ? ? ? ? ?

A B C ? ? ? ? ? C ? ?

slide

j

A B C ? ? ? ? ? C ? ?

$j = k \ (=3)$

length of maximum overlap ABC
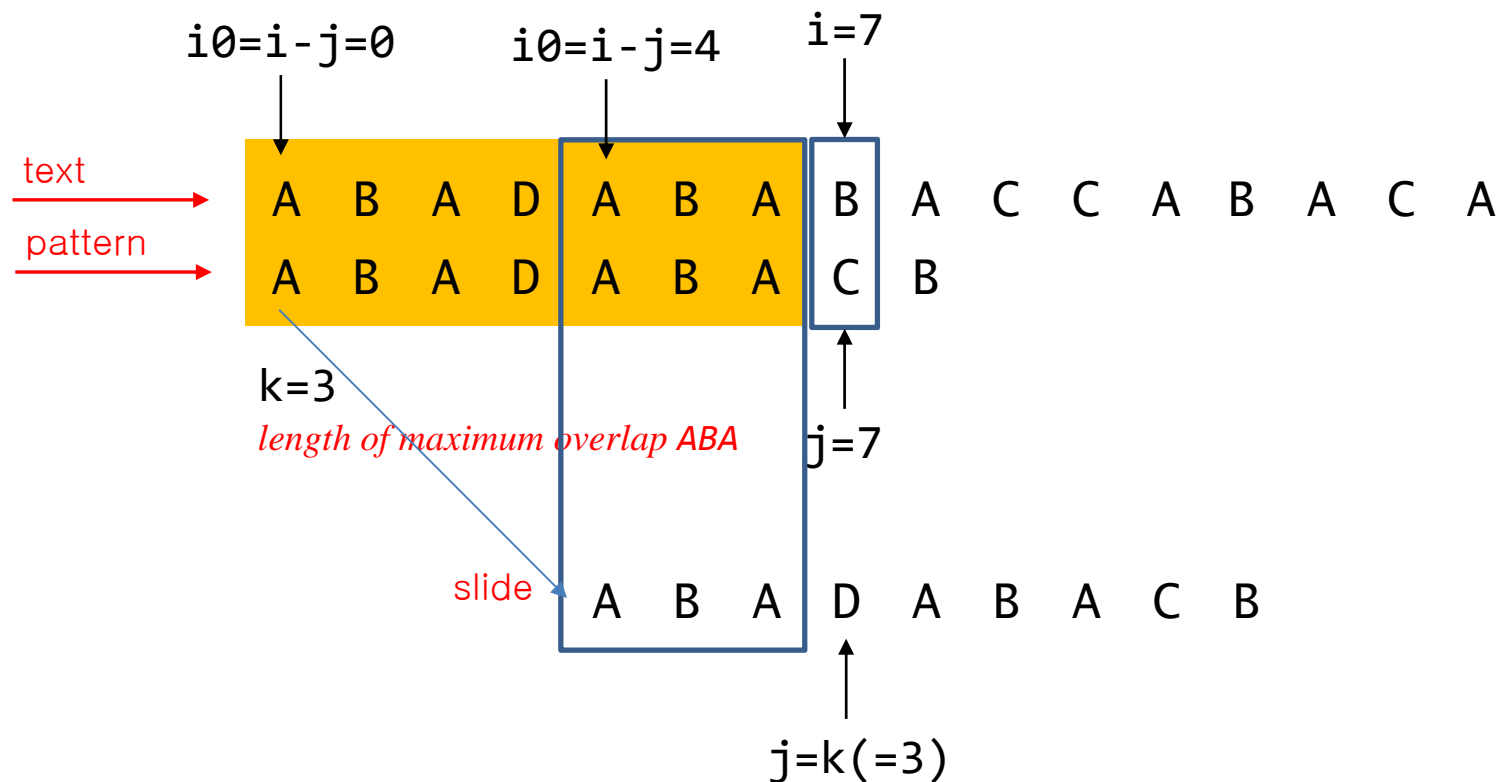
KMP Algorithm

- Fast sliding algorithm:
  - Psuedo program:

```
// mismatch found at text[i], pattern[j]

prefix = pattern[0] ~ pattern[j-1];

k = Length of maximum overlap of prefix;

j = k;
// i is unchanged !

// Matched position i0 in text starts from (i - j);
i0 = i - j;
```

KMP Algorithm

- Fast sliding algorithm:
  - Example:

KMP Algorithm

- Failure function:
  - m : the length of a pattern
  - For $0 < k < m$, the failure function $fail(k)$ is the length of maximum overlap of a prefix `pattern[0]` ~ pattern[k]
    - Note that $fail(0) = 0$

banabana

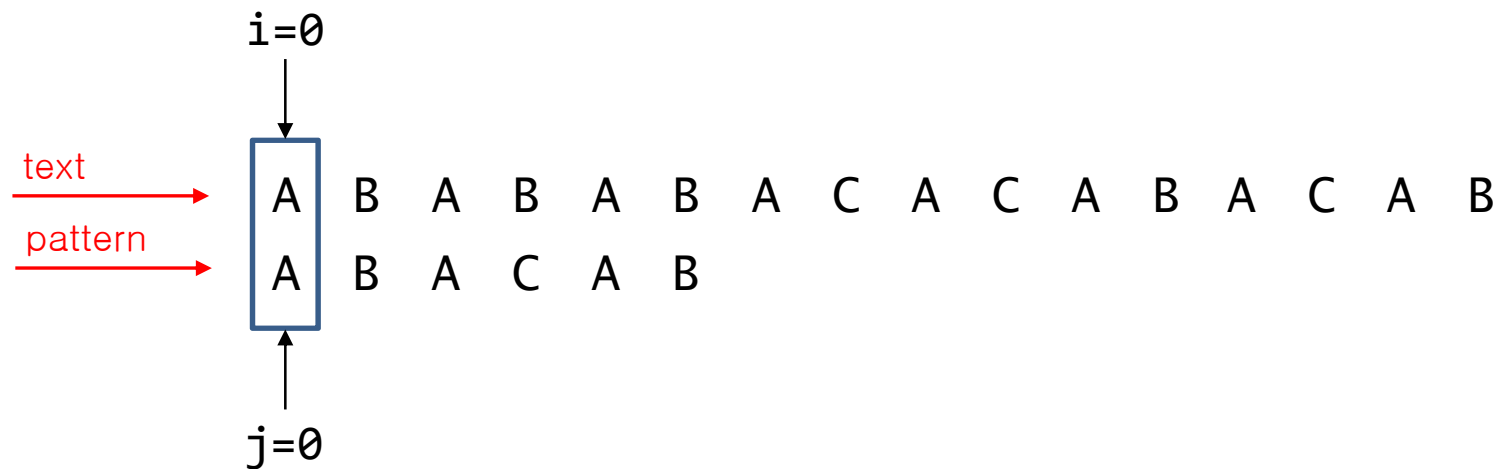| k	prefix	$fail(k)$
0	b	0
1	ba	0
2	ban	0
3	bana	0
4	banab	1
5	banaba	2
6	banaban	0
7	banabana	4

KMP Algorithm

- Knuth-Morris-Pratt(KMP) Algorithm

```cpp
vector<int> kmp(string text, string pattern)
{
    vector<int> ans;
    fail = getFail(pattern);        // failure function
    int n = (int) text.size(), m = (int) pattern.size();
    int j =0;                       // j : index of pattern

    for(int i = 0 ; i < n ; i++)    // i : index of text
    {
        while(j>0 && text[i] != pattern[j])
            j = fail[j-1];
        if(text[i] == pattern[j])
            if(j==m-1)                  // pattern matching is found
            {
                ans.push_back(i-j); // save the matched position
                j = pi[j];
            }
            else
                j++;
    }
    return ans;
}
```
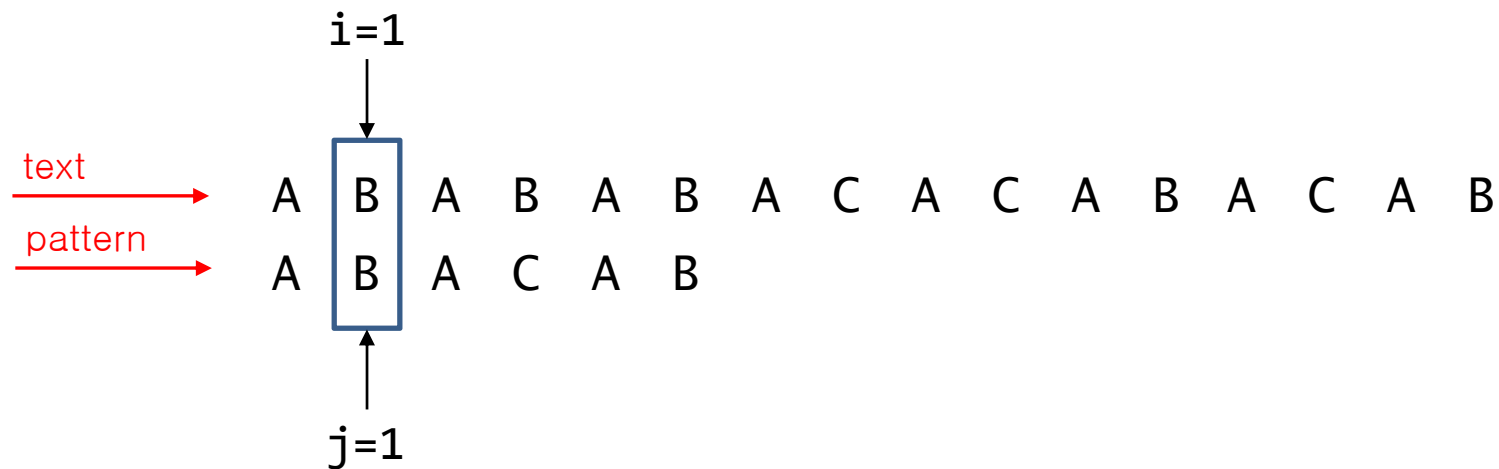
KMP Algorithm

– Example:

i=0

text

pattern

A B A B A B A C A C A B A C A B

A B A C A B

j=0

text[i] == pattern[j]

➔ i++, j++

i	0	1	2	3	4	5
fail(i)	0	0	1	0	1	2

KMP Algorithm

– Example:

i=1

text → A B A B A B A C A C A B A C A B

pattern → A B A C A B

j=1

text[i] == pattern[j]

➔ i++, j++

i	0	1	2	3	4	5
fail(i)	0	0	1	0	1	2

KMP Algorithm

– Example:

i=2

text ⟶ A B [A] B A B A C A C A B A C A B

pattern ⟶ A B [A] C A B

j=2

text[i] == pattern[j]

➔ i++, j++

i	0	1	2	3	4	5
fail(i)	0	0	1	0	1	2

KMP Algorithm

– Example:

i=3

text →   A B A B A B A C A C A B A C A B

pattern →   A B A C A B

j=3

text[i] != pattern[j]

➔ j = fail[j-1]

i	0	1	2	3	4	5
fail(i)	0	0	1	0	1	2

KMP Algorithm

– Example:

i=3

text   A B A B A B A C A C A B A C A B

pattern   A B A C A B

j=1

text[i] == pattern[j]

➔ i++, j++

i	0	1	2	3	4	5
fail(i)	0	0	1	0	1	2

KMP Algorithm

– Example:

i=4

text →

A B A B A B A C A C A B A C A B

pattern →

A B A C A B

j=2

text[i] == pattern[j]

➔ i++, j++

i	0	1	2	3	4	5
fail(i)	0	0	1	0	1	2

KMP Algorithm

– Example:

i=5

text ⟶ A B A B A B A C A C A B A C A B

pattern ⟶ A B A C A B

j=1

text[i] == pattern[j]

➔ i++, j++

i	0	1	2	3	4	5
fail(i)	0	0	1	0	1	2

KMP Algorithm

– Example:

i=9

text

A B A B A B A C A C A B A C A B

pattern

A B A C A B

j=5

text[i] == pattern[j]

➔ i++, j++

i	0	1	2	3	4	5
fail(i)	0	0	1	0	1	2

KMP Algorithm

– Example:

i=9

text → A B A B A B A C A C A B A C A B

pattern → A B A C A B

j=1

text[i] != pattern[j]

➔ j = fail[j-1]

i	0	1	2	3	4	5
fail(i)	0	0	1	0	1	2

KMP Algorithm

– Example:

i=9

text → A B A B A B A C A **C** A B A C A B

pattern → **A** B A C A B

j=0

j==0 && text[i] != pattern[j]

➔ i++

i	0	1	2	3	4	5
fail(i)	0	0	1	0	1	2
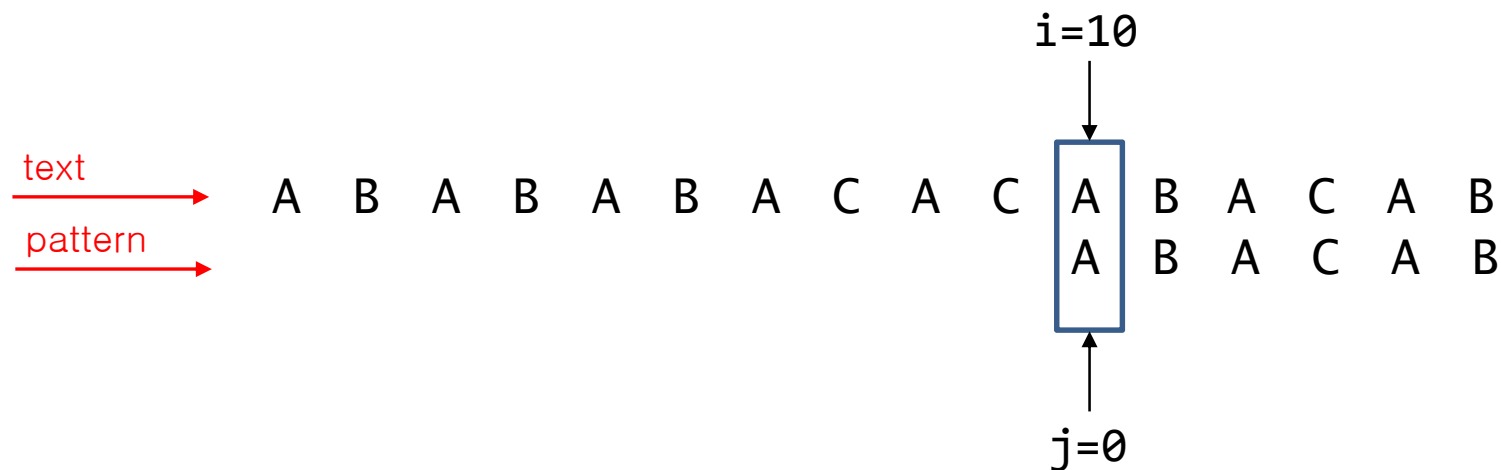
KMP Algorithm

– Example:

i=10

text → A B A B A B A C A C A A B A C A B

pattern → A B A C A B

j=0

text[i] == pattern[j]

➔ i++, j++

i	0	1	2	3	4	5
fail(i)	0	0	1	0	1	2

KMP Algorithm

– Example:

i0=i-j=10

i=15

text → A B A B A B A C A C A B A C A B

pattern → A B A C A B

j=5

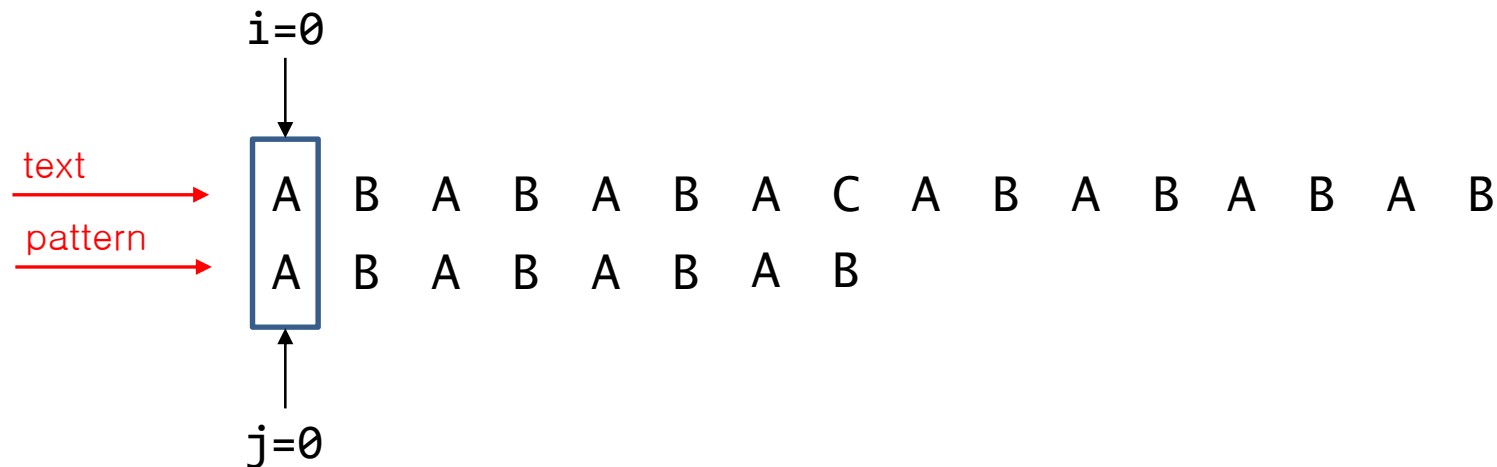| i	0	1	2	3	4	5
fail(i)	0	0	1	0	1	2

KMP Algorithm

- Knuth-Morris-Pratt(KMP) Algorithm
  - Why?

```
while(j>0 && text[i] != pattern[j])
        j = pi[j-1];
```

i=0

text → A B A B A B A C A B A B A B A B

pattern → A B A B A B A B

j=0
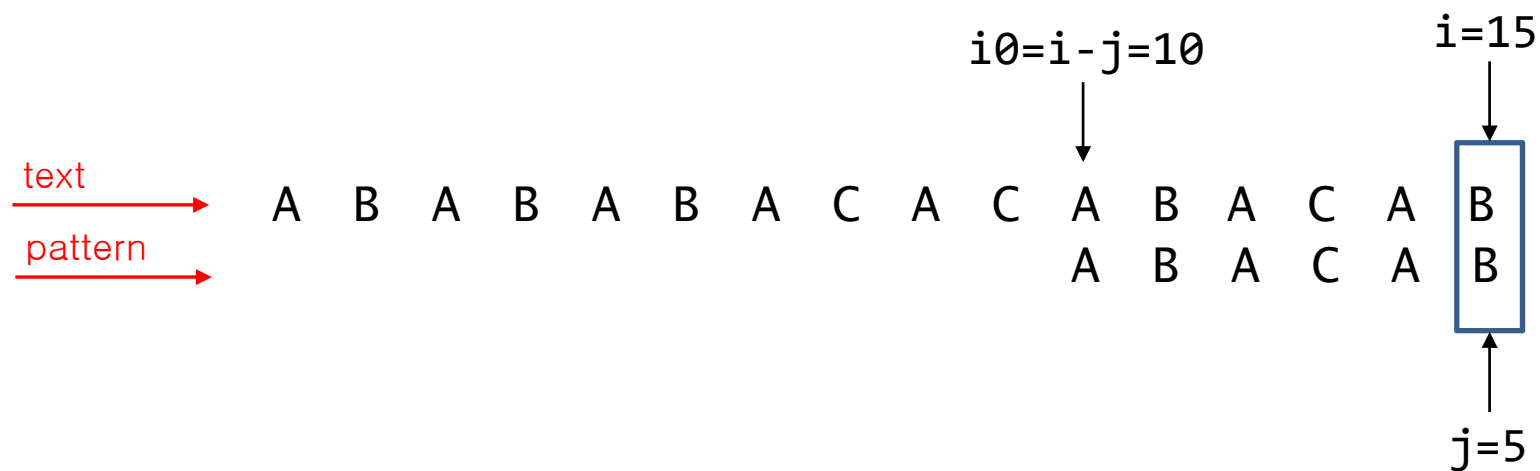
text[i] == pattern[j]

➔ i++, j++

i	0	1	2	3	4	5	6	7
fail(i)	0	0	1	2	3	4	5	0

KMP Algorithm

- Knuth-Morris-Pratt(KMP) Algorithm
  - Why?

```
while(j>0 && text[i] != pattern[j])
        j = pi[j-1];
```
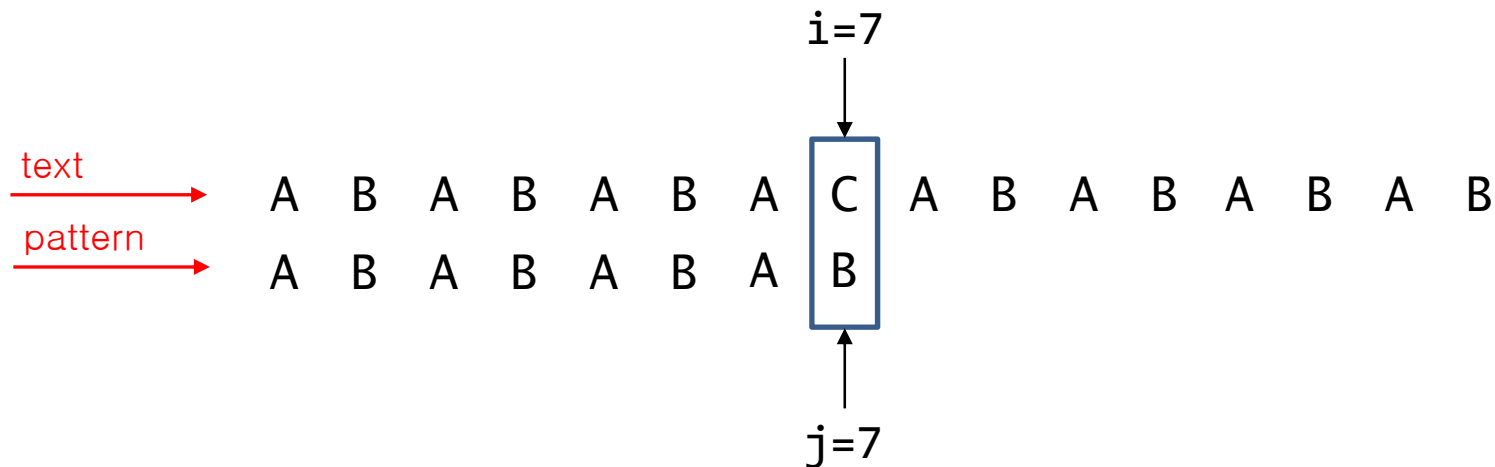
i=7

text → A B A B A B A C A B A B A B A B

pattern → A B A B A B A B

j=7

text[i] != pattern[j]

➔ j = fail[j-1]

i	0	1	2	3	4	5	6	7
fail(i)	0	0	1	2	3	4	5	0

KMP Algorithm

- Knuth-Morris-Pratt(KMP) Algorithm
  - Why?

```
while(j>0 && text[i] != pattern[j])
        j = pi[j-1];
```
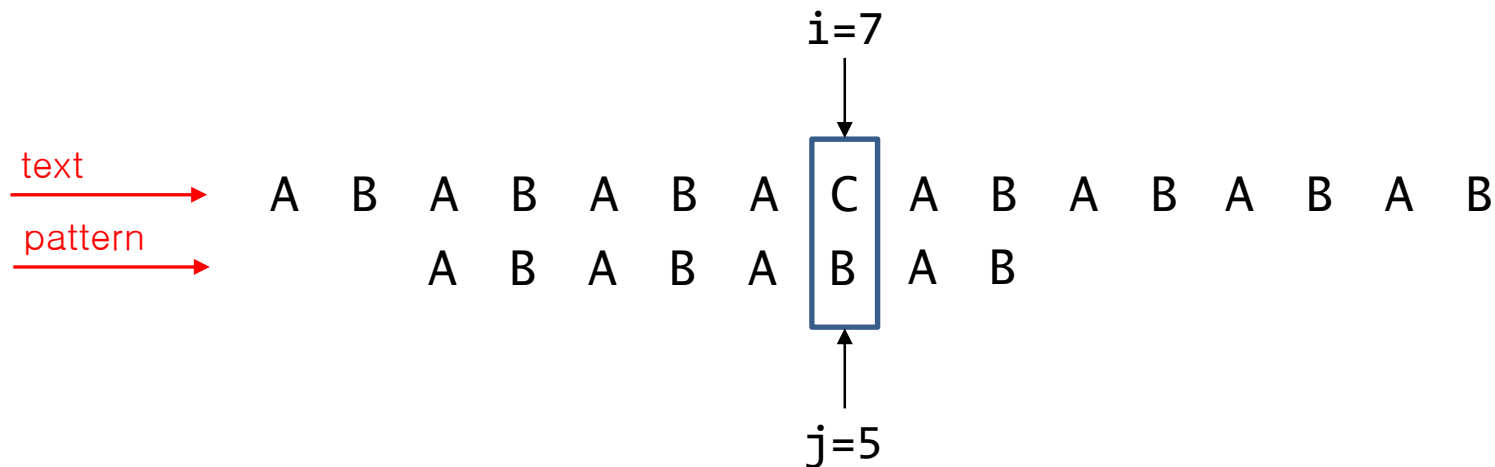
i=7

text →

A B A B A B A C A B A B A B A B

pattern →

A B A B A B A B

j=5

text[i] != pattern[j]

→ j = fail[j-1]

i	0	1	2	3	4	5	6	7
fail(i)	0	0	1	2	3	4	5	0

KMP Algorithm

- Knuth-Morris-Pratt(KMP) Algorithm
  - Why?

```
while(j>0 && text[i] != pattern[j])
          j = pi[j-1];
```

i=7

text → A B A B A B A C A B A B A B A B
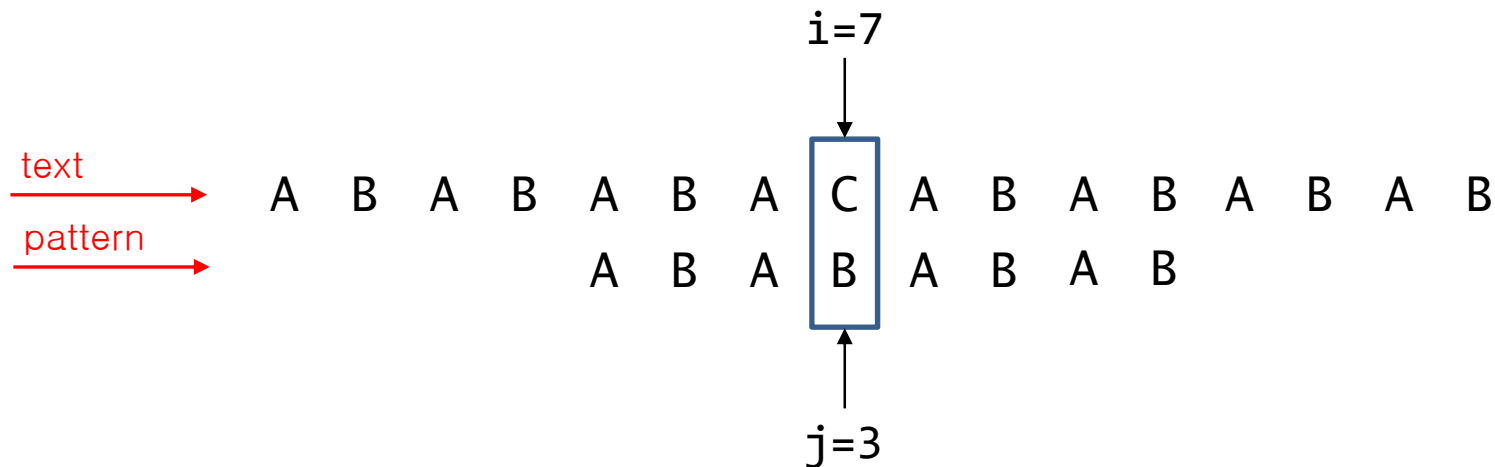
pattern → A B A B A B A B

j=3

text[i] != pattern[j]

➔ j = fail[j-1]

i	0	1	2	3	4	5	6	7
fail(i)	0	0	1	2	3	4	5	0

KMP Algorithm

- Knuth-Morris-Pratt(KMP) Algorithm
  - Why?

```
while(j>0 && text[i] != pattern[j])
         j = pi[j-1];
```
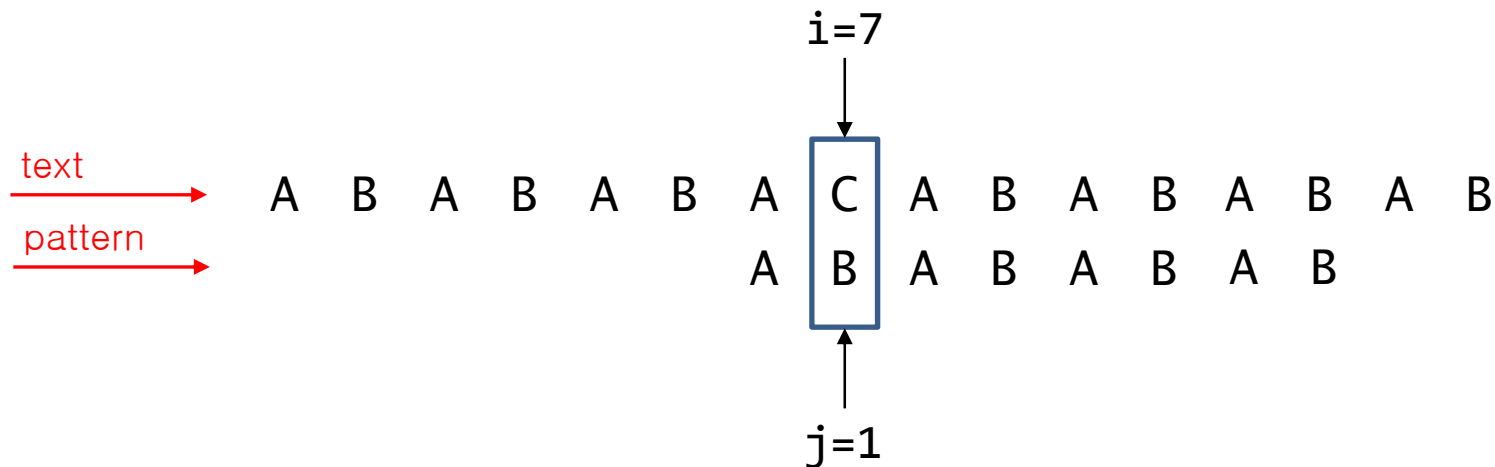
i=7

text →

A B A B A B A C A B A B A B A B

pattern →

A B A B A B A B

j=1

text[i] != pattern[j]

➔ j = fail[j-1]

i	0	1	2	3	4	5	6	7
fail(i)	0	0	1	2	3	4	5	0

KMP Algorithm

- Knuth-Morris-Pratt(KMP) Algorithm
  - Why?

```
while(j>0 && text[i] != pattern[j])
        j = pi[j-1];
```
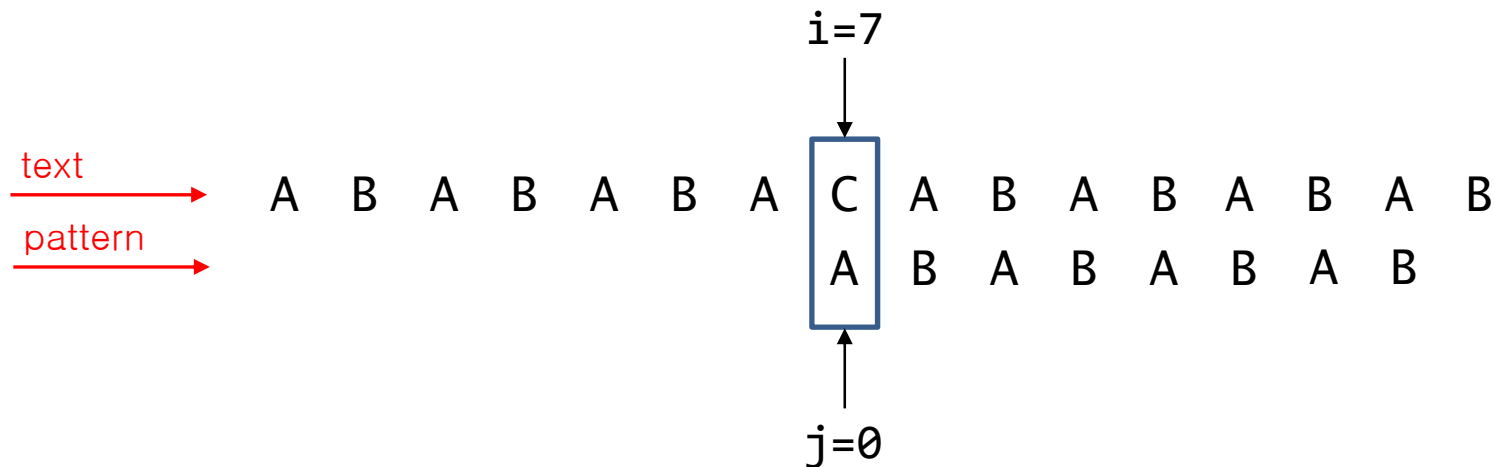
i=7

text → A B A B A B A **C** A B A B A B A B

pattern → **A** B A B A B A B

j=0

text[i] != pattern[j]

➔ j = fail[j-1]

i	0	1	2	3	4	5	6	7
fail(i)	0	0	1	2	3	4	5	0

- Knuth-Morris-Pratt(KMP) Algorithm
  - Why?

```
while(j>0 && text[i] != pattern[j])
        j = pi[j-1];
```

i=8

text →

pattern →

A B A B A B A C A B A B A B A B
                A B A B A B A B

j=0

j==0 && text[i] != pattern[j]

➔ i++

i	0	1	2	3	4	5	6	7
fail(i)	0	0	1	2	3	4	5	0