Lecture #20
String Matching (3)

Algorithm

JBNU

Jinhong Jung

In This Lecture

- More efficient algorithm for string matching
  - KMP algorithm

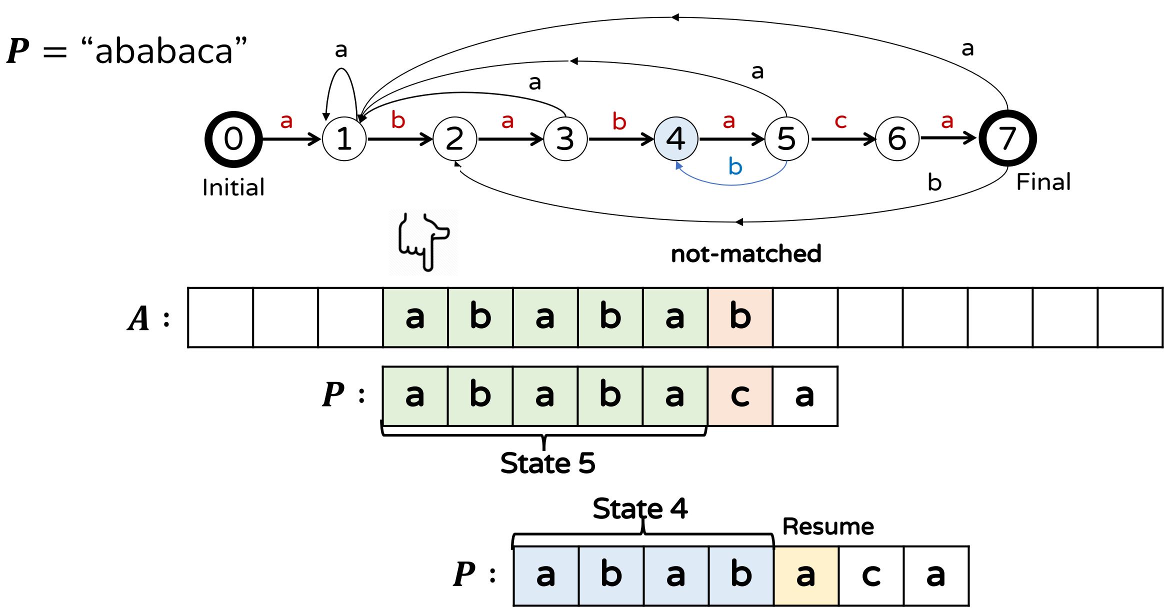
Outline

- Intuition of KMP algorithm
- Search phase
- Failure array construction phase

Remind String Mating Automata

❑ Where does the efficiency of automata come from?

- When a match fails, the automata knows where we go back and resume matching ⇒ don't need to match from scratch

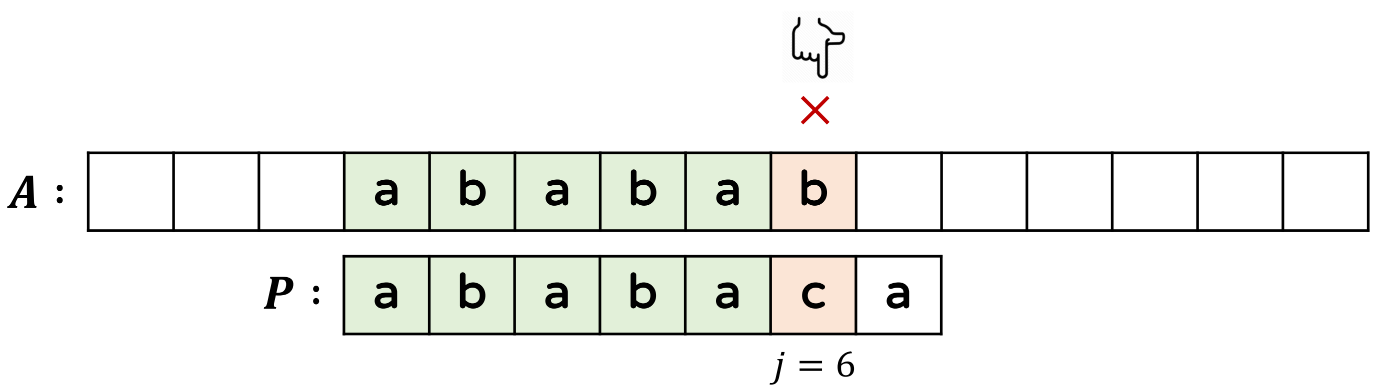


T	a	b	c	*
0	1	0	0	0
1	1	2	0	0
2	1	0	0	0
3	1	4	0	0
4	5	0	0	0
5	1	4	6	0
6	7	0	0	0
7	1	2	0	0

- But, it takes $O(|\mathbf{\Sigma}|m)$ space & $O(|\mathbf{\Sigma}|m^3)$ time for construction
  - Can we do better? How to remove Σ?

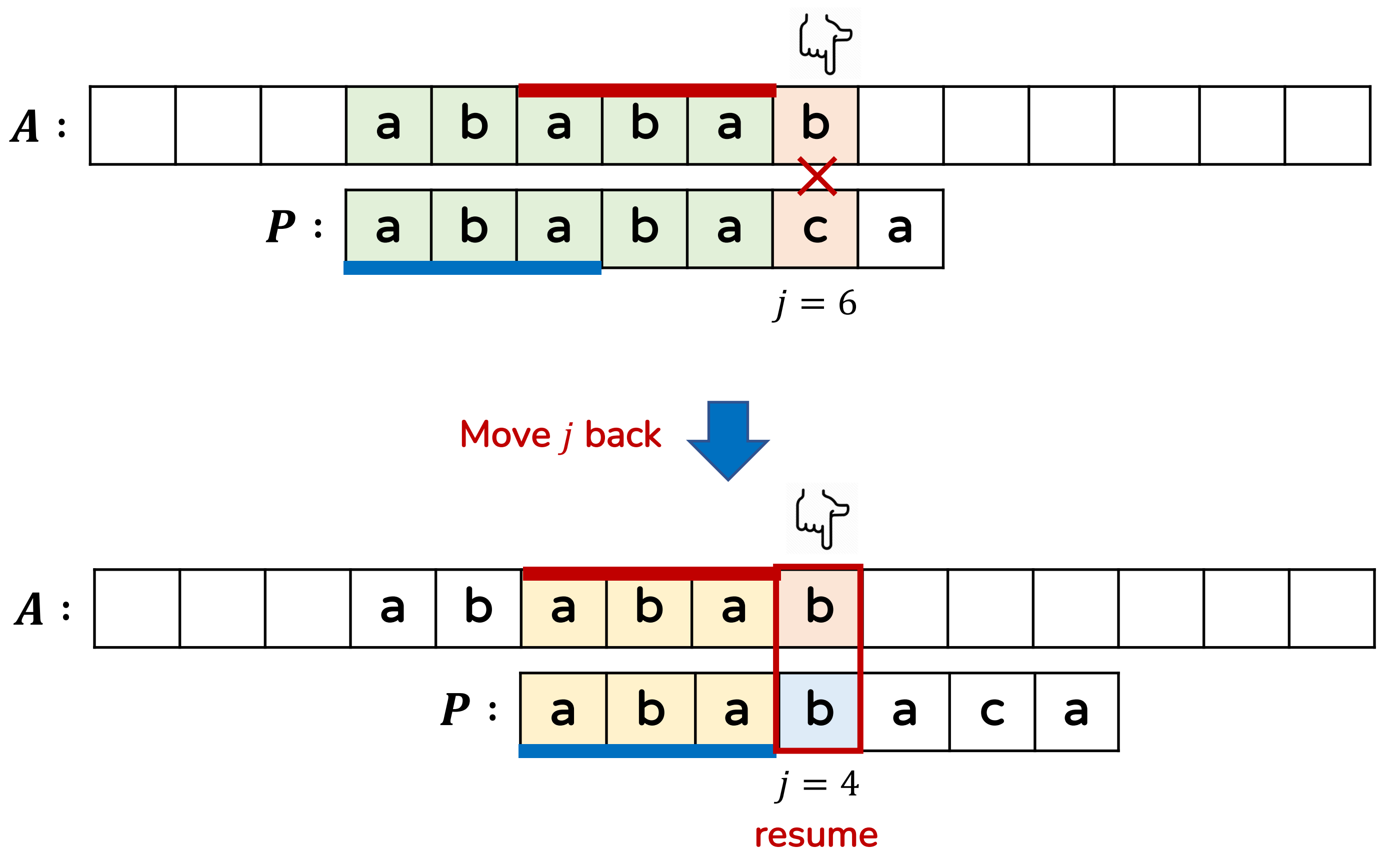
Intuition of KMP Algorithm (1)

- Let's introduce a failure symbol ($\times$) instead of Σ to indicate that a match fails
  - For example, a match fails at $j = 6$; then, the automata handle this event with "b"
  - Instead of this, let's handle this with a single symbol $\times$



Intuition of KMP Algorithm (2)

- To handle ×, we can use the LPS of "**ababa**" for the next match (here, the LPS is "aba")
- Equal to moving j to 4 (**next to LPS**); then, resume matching!



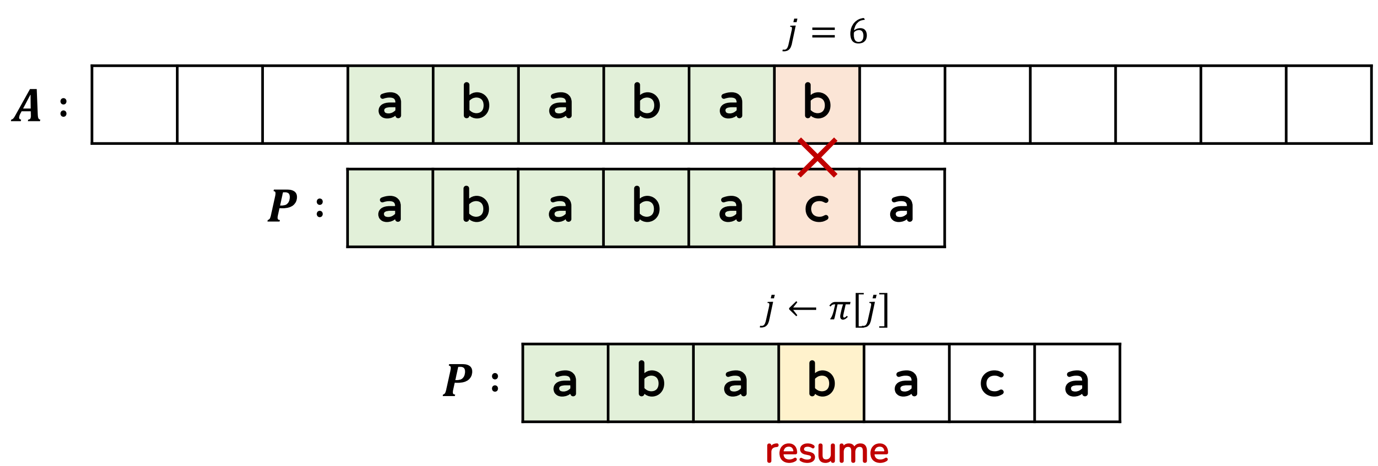
Failure Array π

❑ Contains the information on how many we go back to when a match fail (×)

- For example, P =“abababaca” results in the following
  - $\pi[j]$ indicates a resuming location in P when a match fails (×)
  - $\pi[j]$ = 1 + length of LPS of $P[1 \cdots j-1]$

$\pi =$

0	1	2	3	4	5	6	7	8
×	0	1	1	2	3	4	1	2

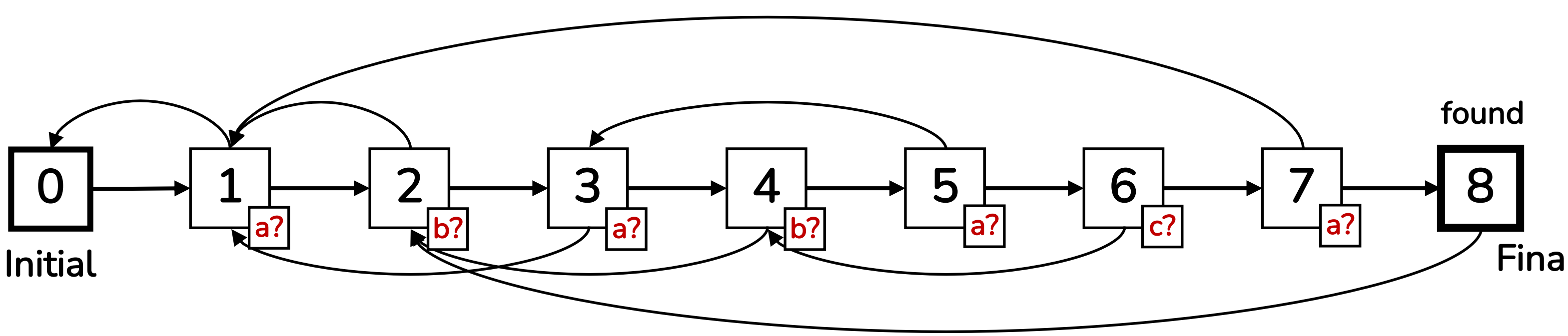


Failure Automata

❑ π represents the following failure automata

- Every backward edge indicates failure matching (×)
- A note indicates a state; after we visit the state, we should compare a character in the small box
- It uses $O(m)$ extra space! (we'll see how to construct $\boldsymbol{\pi}$ later)

	0	1	2	3	4	5	6	7	8
$\boldsymbol{\pi} =$	×	0	1	1	2	3	4	1	2



Overview of KMP Algorithm

❑ Proposed by Knuth, Morris, and Pratt in 1977

- Has a similar intuition to that of string matching automata
  - Restart from a resuming location when a match fails, not from scratch

❑ Phases of KMP Algorithm

- **Failure array construction phase:** Construct $\boldsymbol{\pi}$ from $\boldsymbol{P}$
- **Search phase:** Match $\boldsymbol{P}$ over $\boldsymbol{A}$ with $\boldsymbol{\pi}$
  - Let's first check the search phase assuming a valid $\boldsymbol{\pi}$ is given.
  - Correctness is out-of-scope. Instead, focus on the intuition!
  - Note that there are various implementations of KMP according to interpretation and index base.
    - This lecture shows the simplest version using 1-base index, included in the textbook.

Outline

- ❑ Intuition of KMP algorithm
- ❑ Search phase
- ❑ Failure array construction phase

Search Phase of KMP

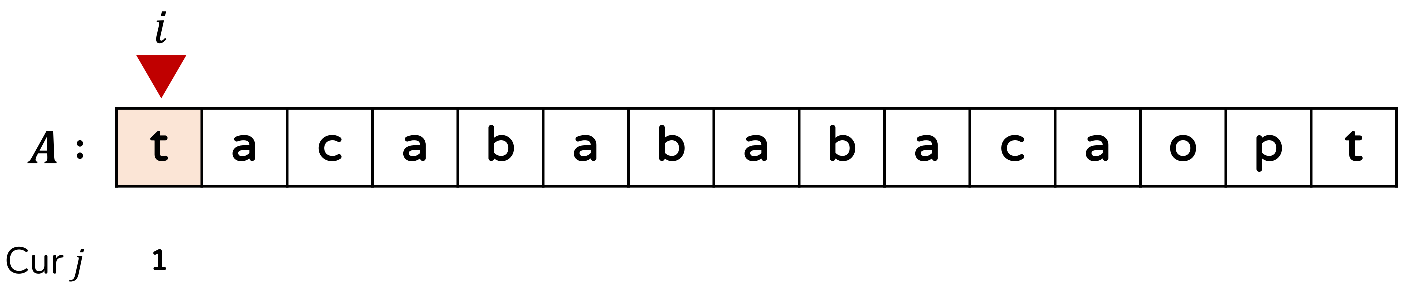
❑ Overview of the search phase

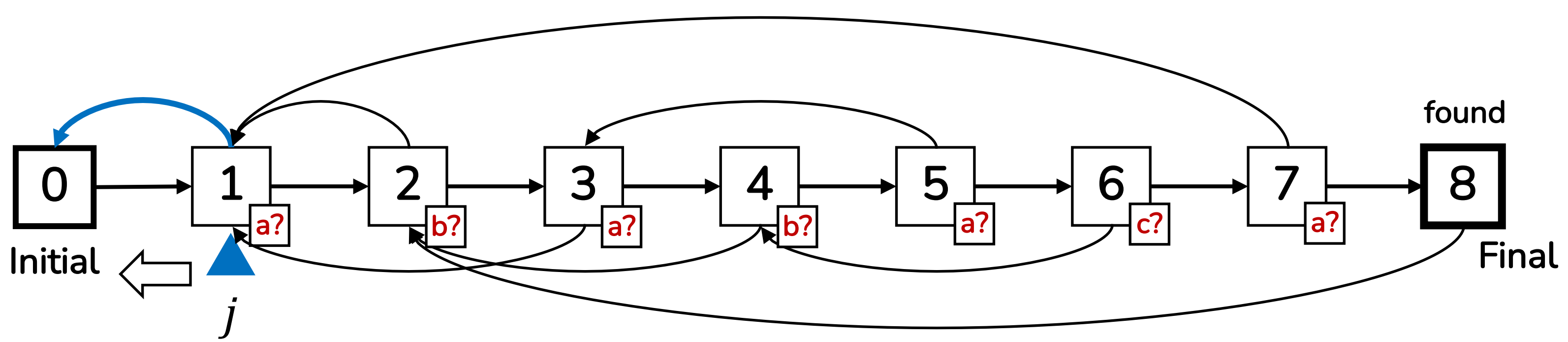
- Input: $\boldsymbol{A}$, $\boldsymbol{P}$, and $\boldsymbol{\pi}$
  - i is a variable pointing to $\boldsymbol{A}$ and j is a variable point to $\boldsymbol{P}$
- While sequentially iterating $\boldsymbol{A}$ from left to right, handle the following cases:
  - Initial or match case
    - If $j = 0$ or $\boldsymbol{A}[i] = \boldsymbol{P}[j]$, then move i and j to the next ($i \leftarrow i + 1$ and $j \leftarrow j + 1$)
  - Failure case
    - If $\boldsymbol{A}[i] \neq \boldsymbol{P}[j]$, then go back to $j \leftarrow \pi[j]$
  - Final case
    - If $j = m + 1$, then output that $\boldsymbol{P}$ is matched at $A[i - m]$ and go back to $j \leftarrow \pi[j]$

Search Phase with π (1)

P="ababaca"

- Start at $A[1]$ & State 1, and compare $A[i]$ & $P[j]$
  - $\Rightarrow$ Failure! Move j back ($j \leftarrow \pi[j]$)





Search Phase with π (2)

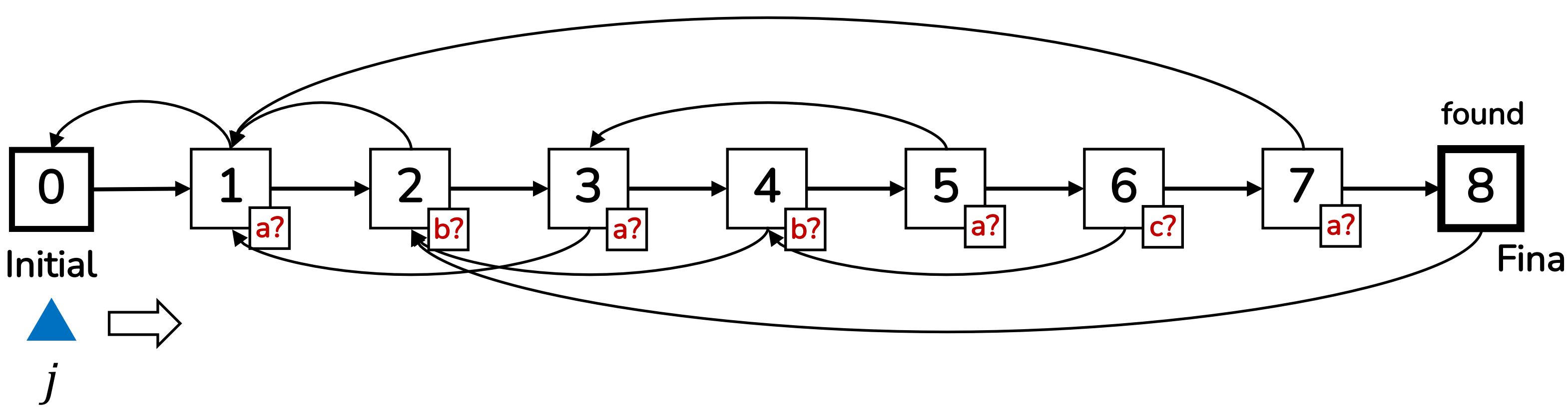
P="ababaca"

❑ This is the initial case ($j = 0$)

- Then, move i and j to the next

i

A :	t	a	c	a	b	a	b	a	b	a	c	a	o	p	t
	1														
Cur j	0														

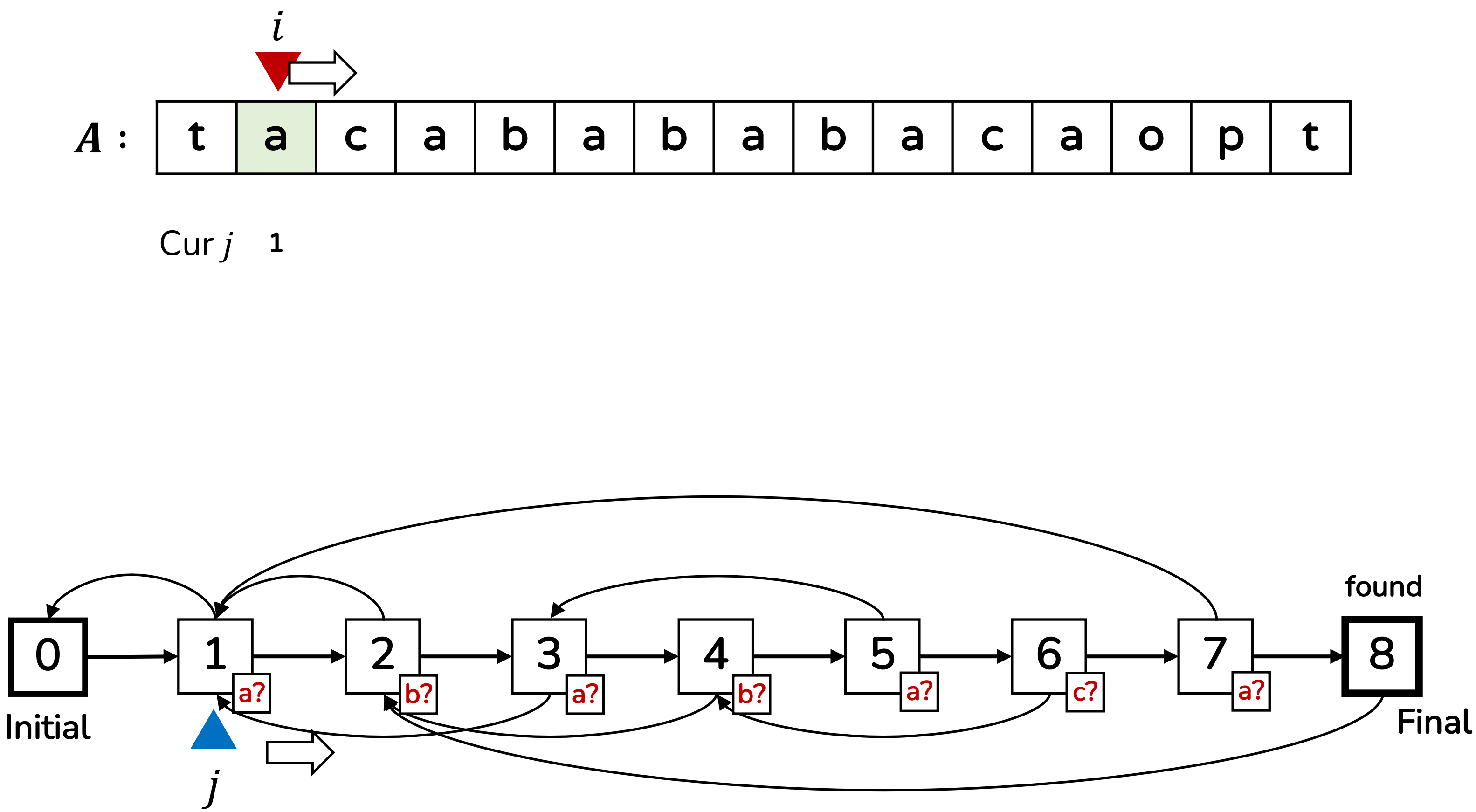


Search Phase with π (3)

P="ababaca"

❑ Compare $A[i]$ and $P[j]$

- ⇒ Match! Move i and j to the next

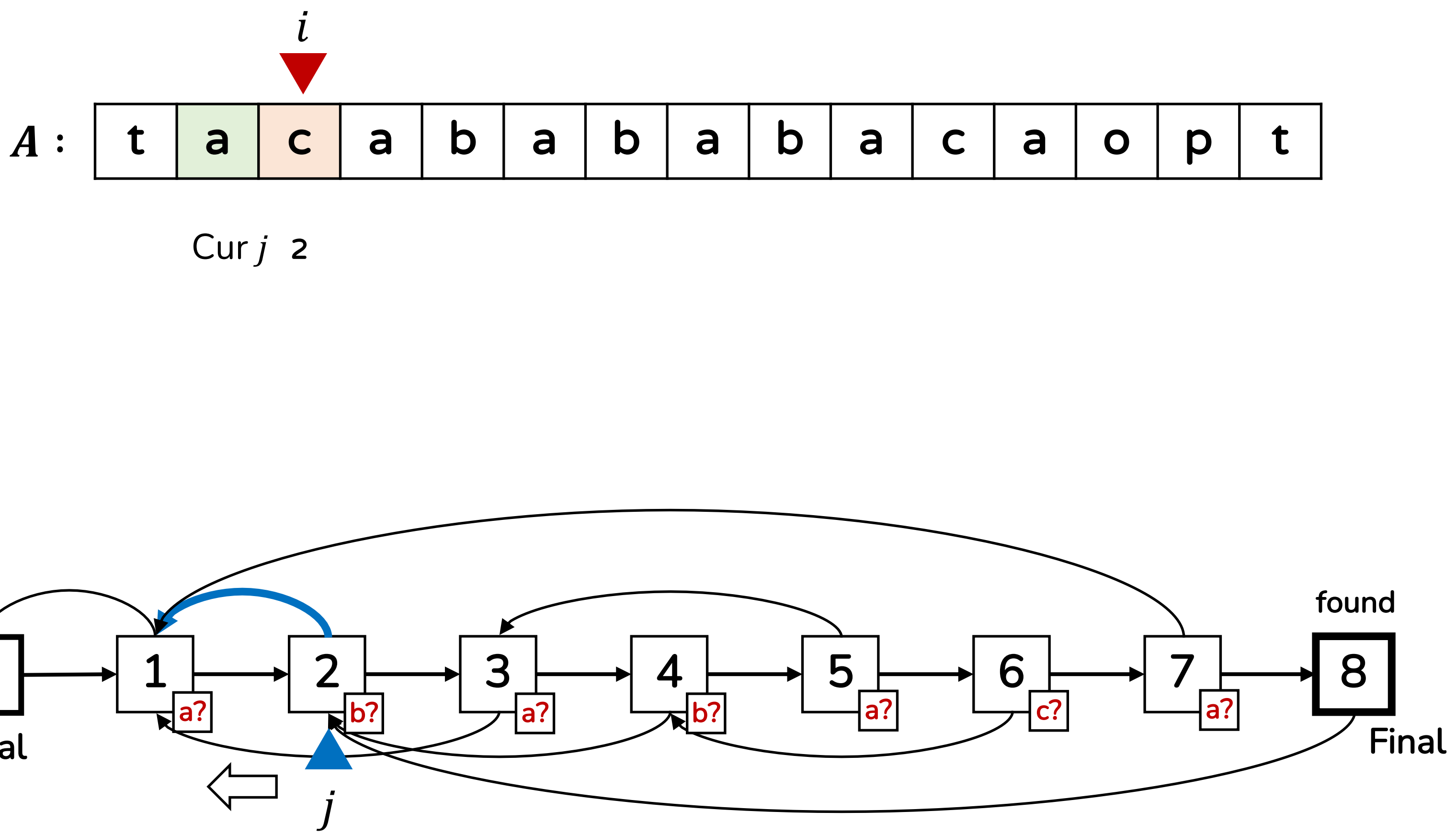


Search Phase with π (4)

P=“ababaca”

❑ Compare $A[i]$ and $P[j]$

- ⇒ Failure! Move j back ($j \leftarrow \pi[j]$)

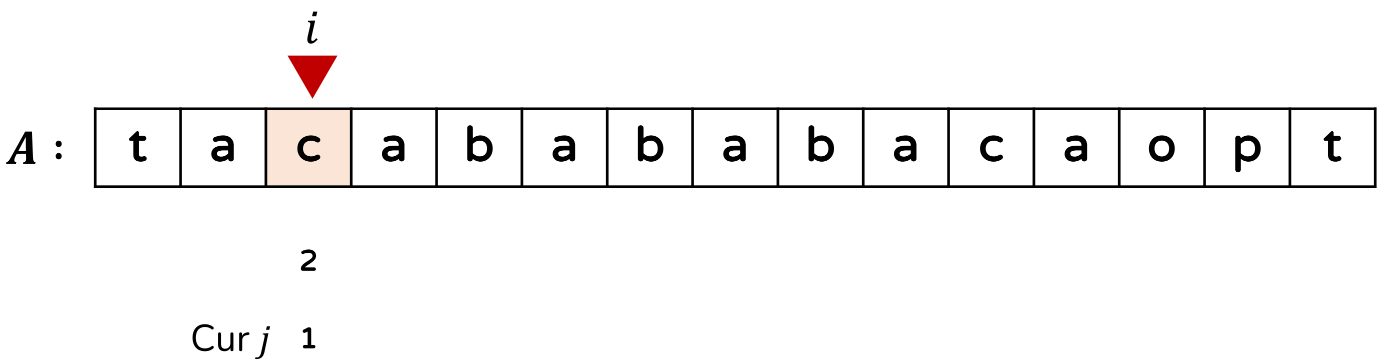


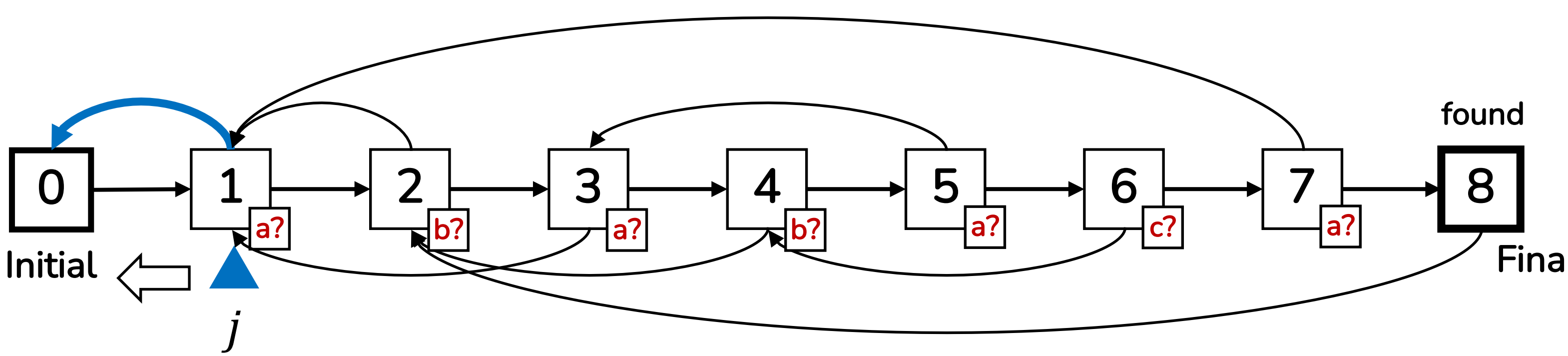
Search Phase with π (5)

P="ababaca"

❑ Compare $A[i]$ and $P[j]$

- ⇒ Failure! Move j back ($j \leftarrow \pi[j]$)



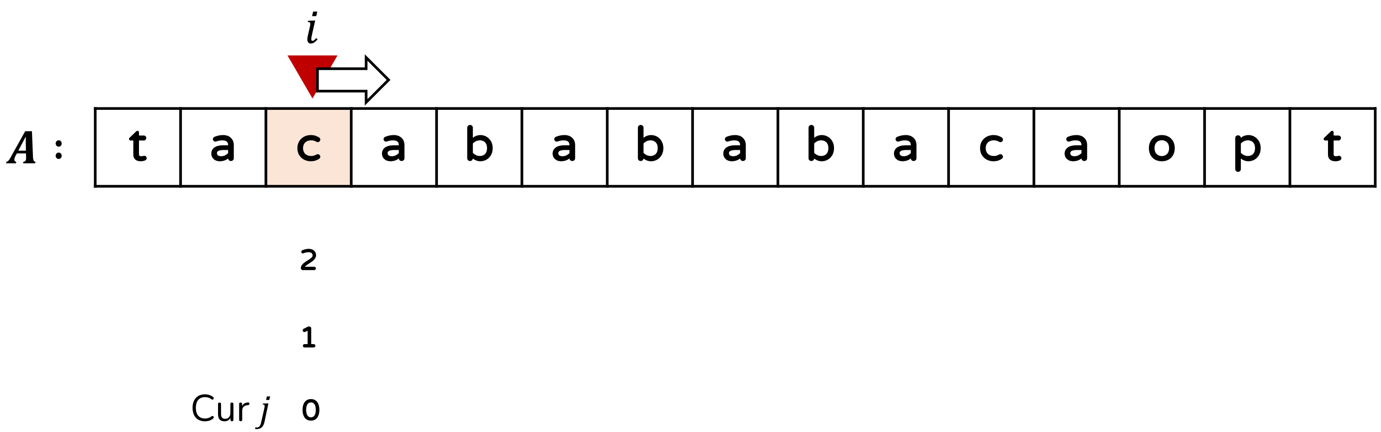


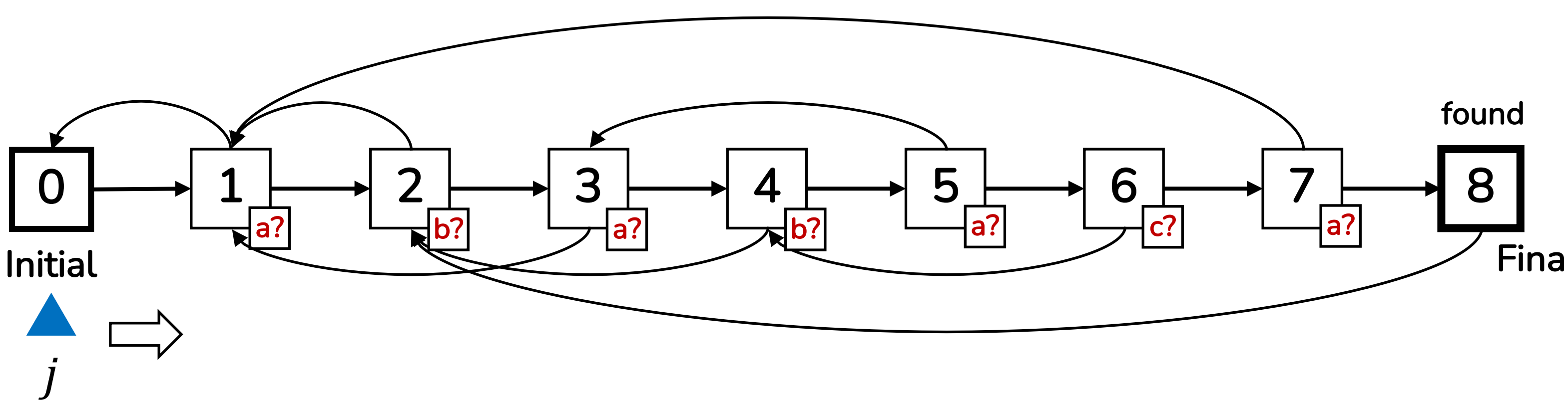
Search Phase with π (6)

P="ababaca"

❑ This is the initial case ($j = 0$)

- Then, move i and j to the next



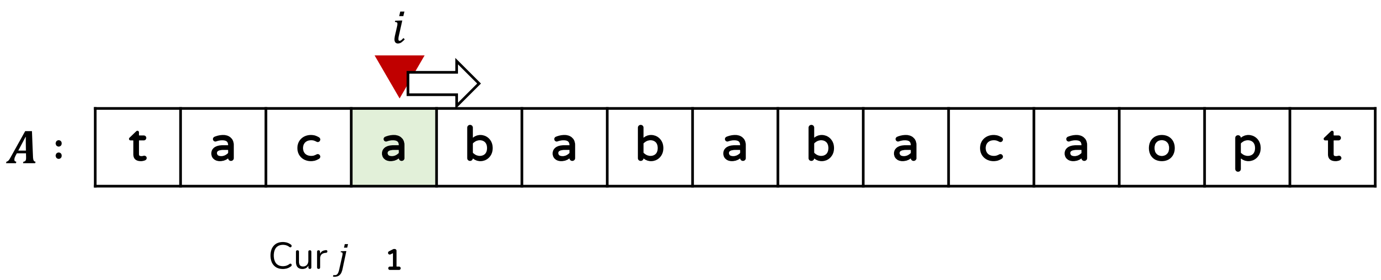


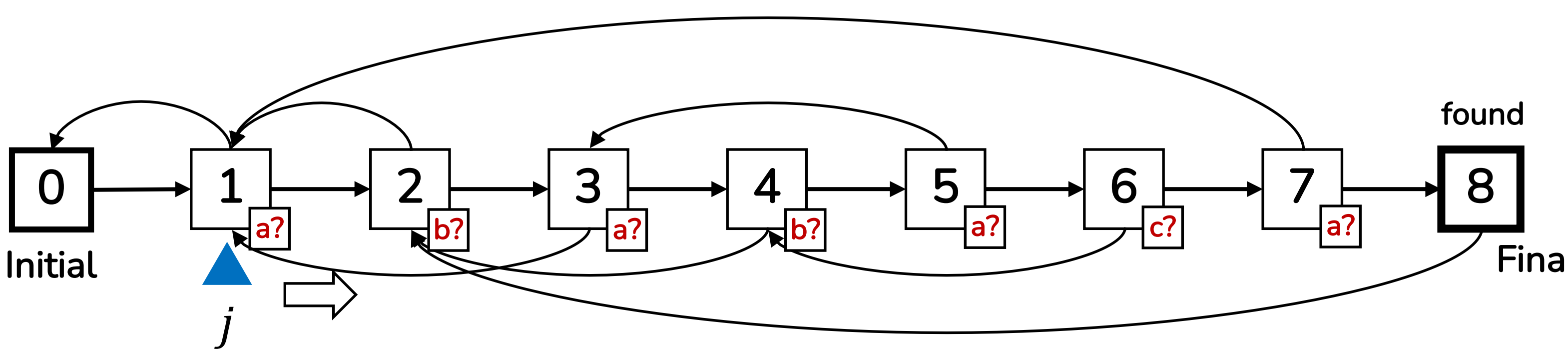
Search Phase with π (7)

P="ababaca"

❑ Compare $A[i]$ and $P[j]$

- ⇒ Match! Move i and j to the next



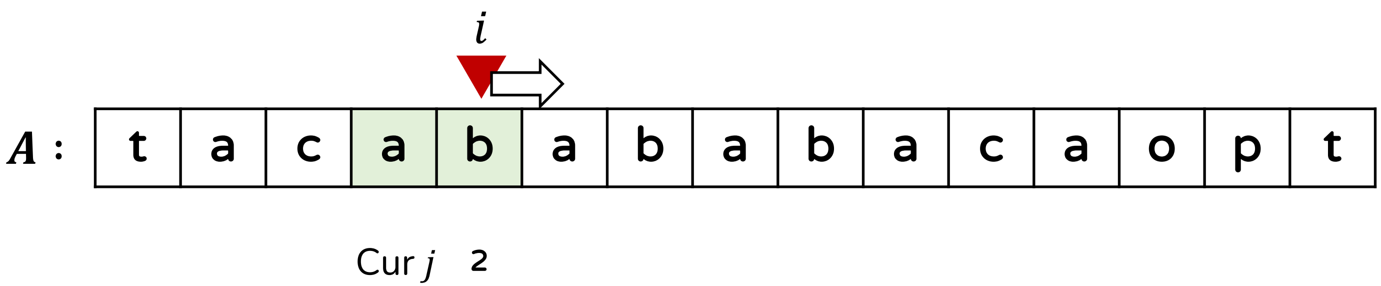


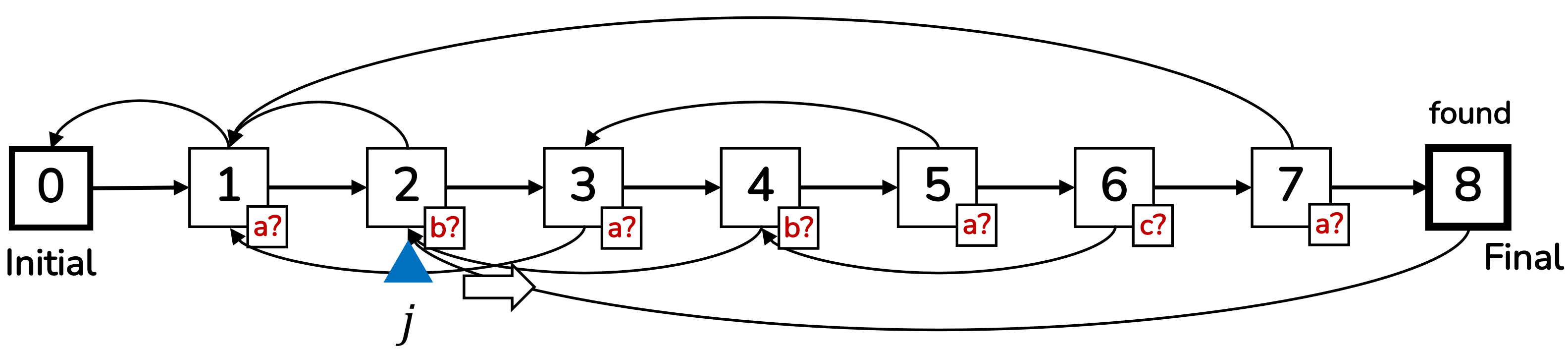
Search Phase with π (8)

P=“ababaca”

❑ Compare $A[i]$ and $P[j]$

- ⇒ Match! Move i and j to the next



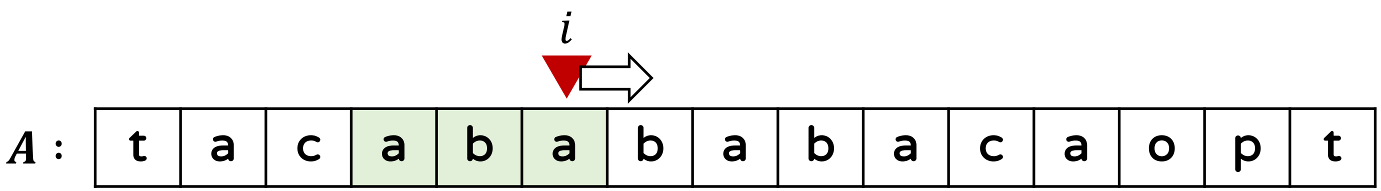


Search Phase with π (9)

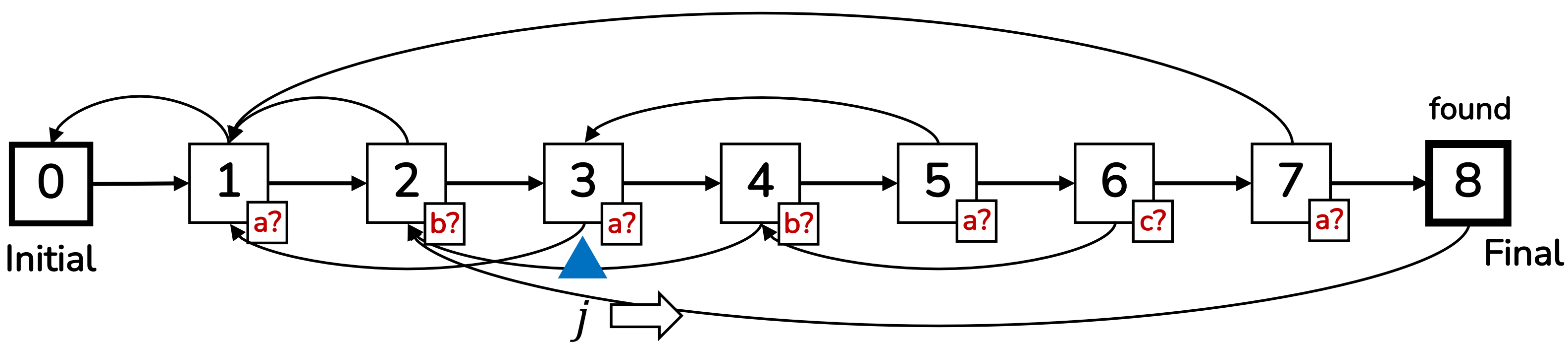
P="ababaca"

❑ Compare $A[i]$ and $P[j]$

- ⇒ Match! Move i and j to the next



Cur j 3

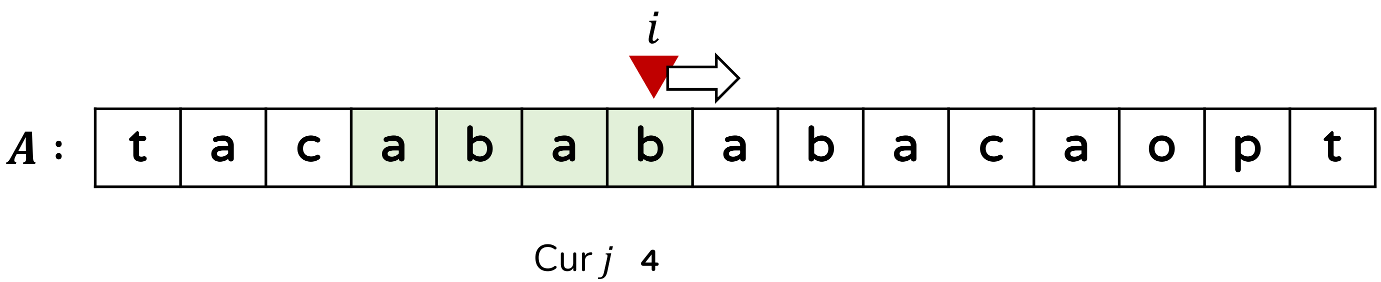


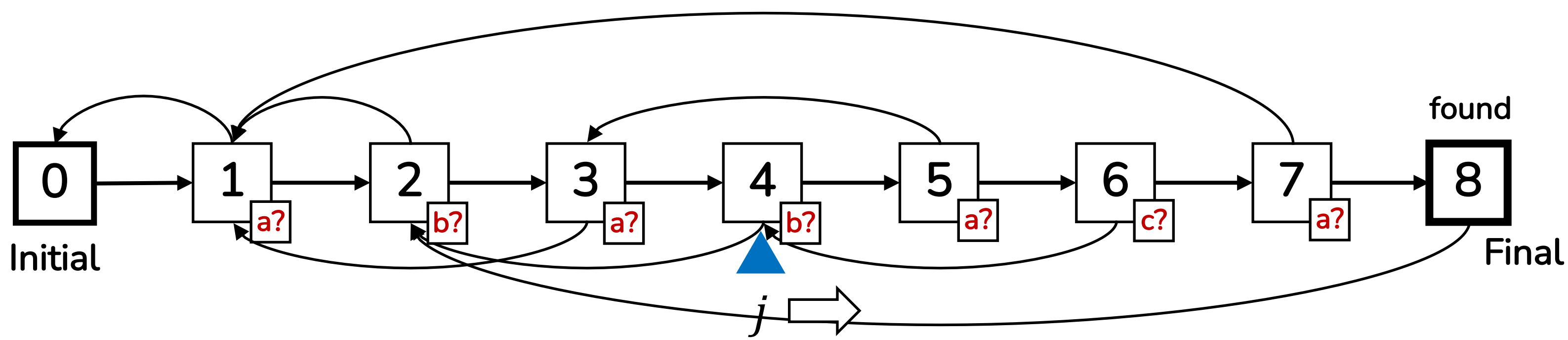
Search Phase with π (10)

P="ababaca"

❑ Compare $A[i]$ and $P[j]$

- ⇒ Match! Move i and j to the next



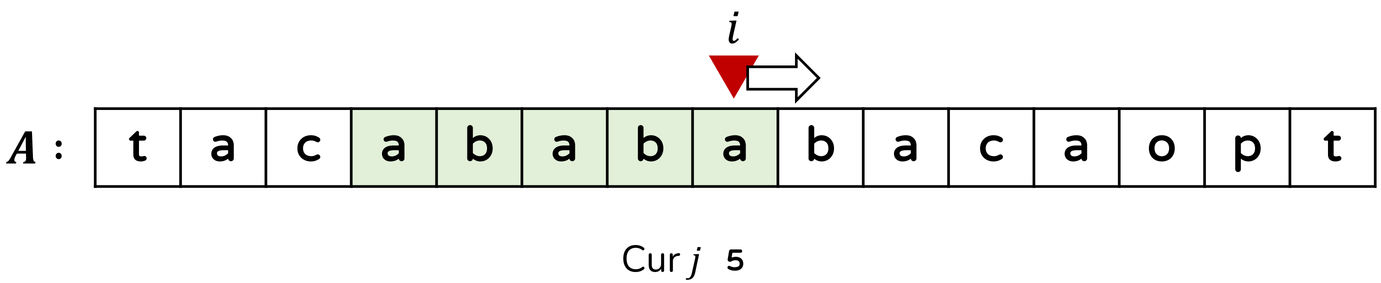


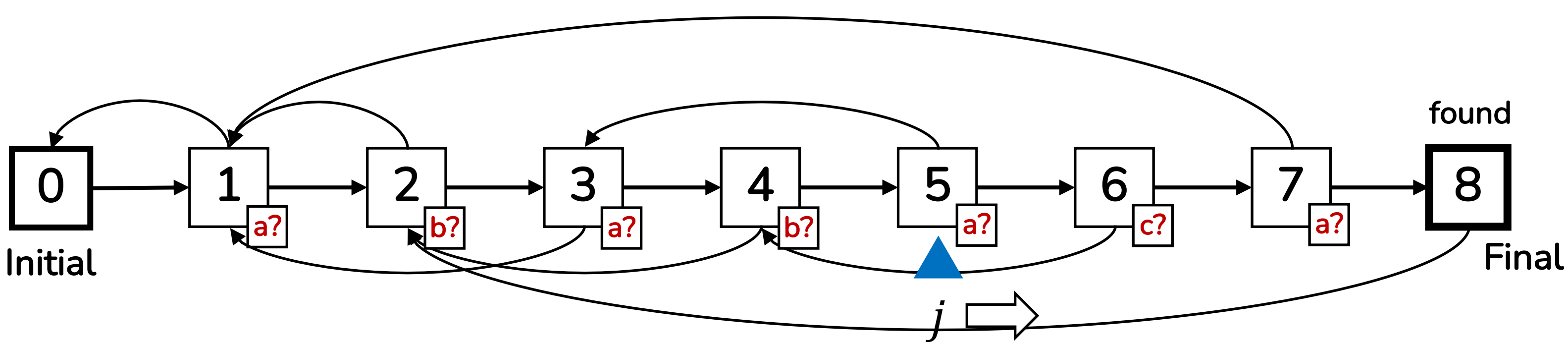
Search Phase with π (11)

P="ababaca"

❑ Compare $A[i]$ and $P[j]$

- ⇒ Match! Move i and j to the next



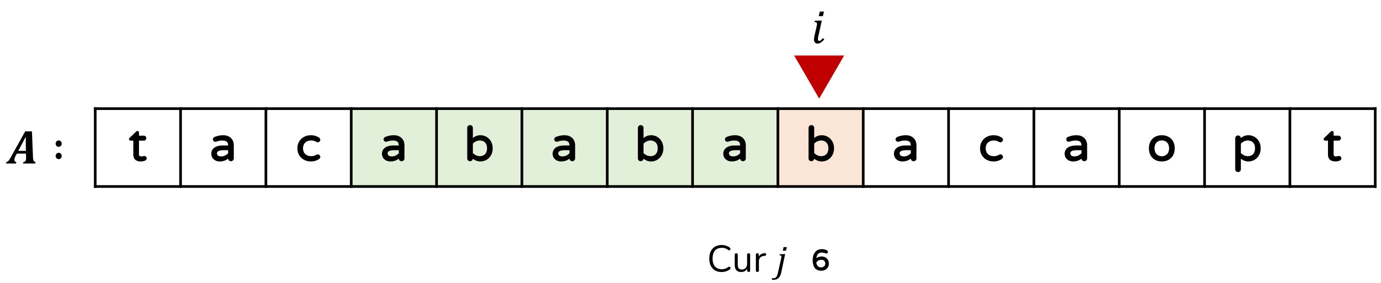


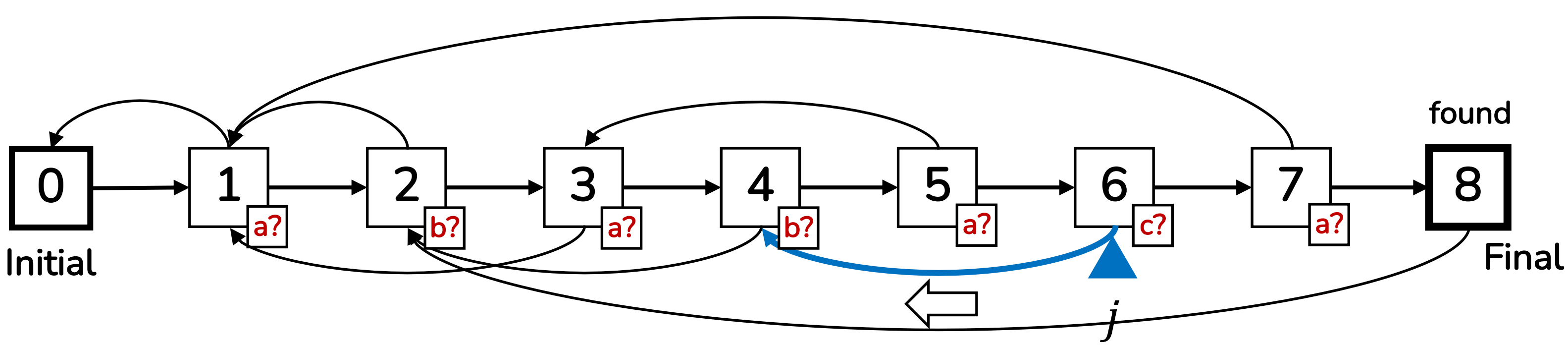
Search Phase with π (12)

P="ababaca"

❑ Compare $A[i]$ and $P[j]$

- ⇒ Failure! Move j back ($j \leftarrow \pi[j]$)



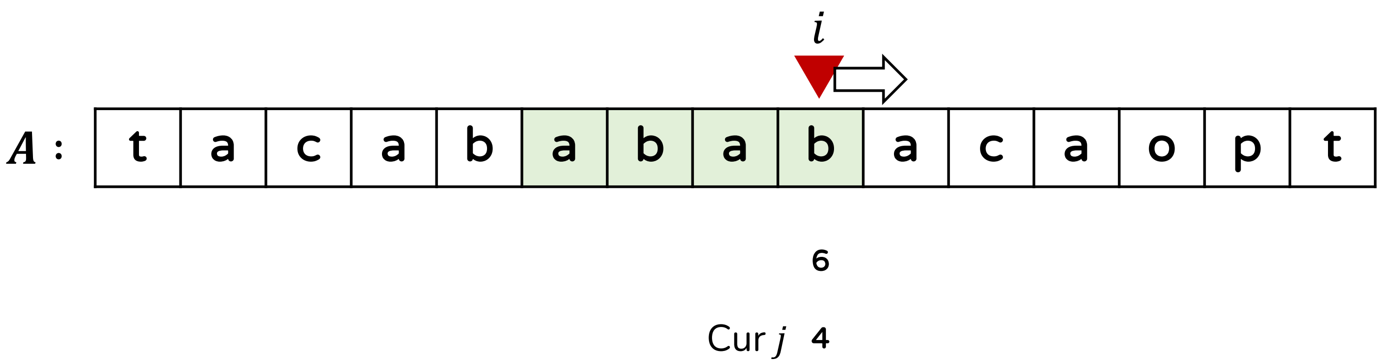


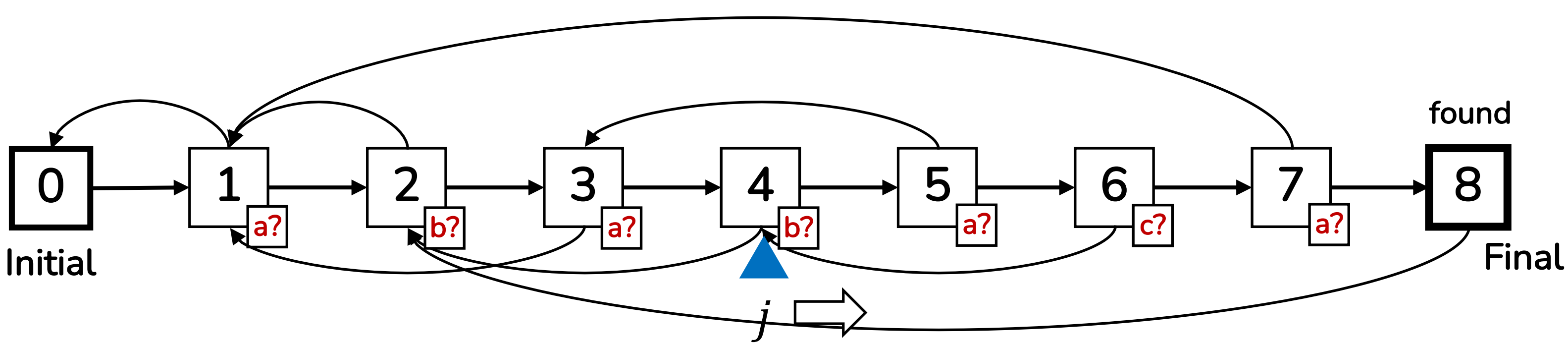
Search Phase with π (13)

P="ababaca"

❑ Compare $A[i]$ and $P[j]$

- ⇒ Match! Move i and j to the next



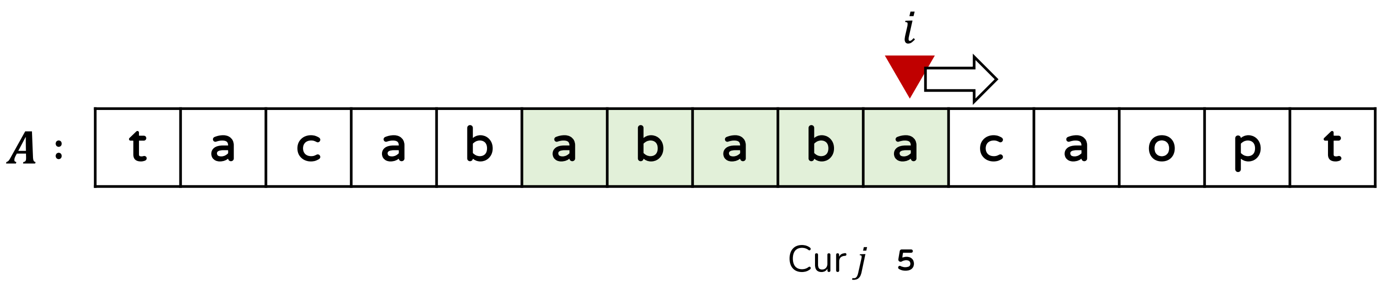


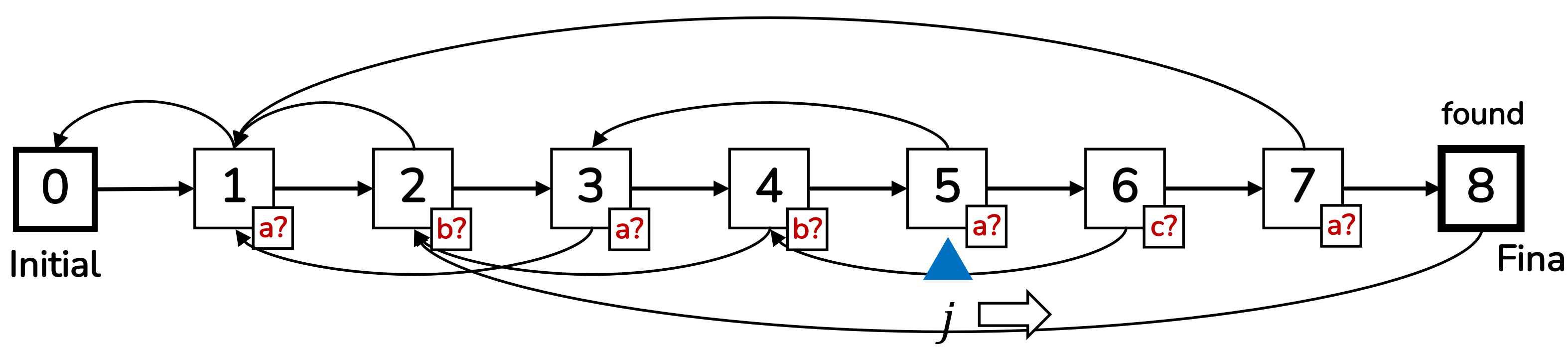
Search Phase with π (14)

P="ababaca"

❑ Compare $A[i]$ and $P[j]$

- ⇒ Match! Move i and j to the next



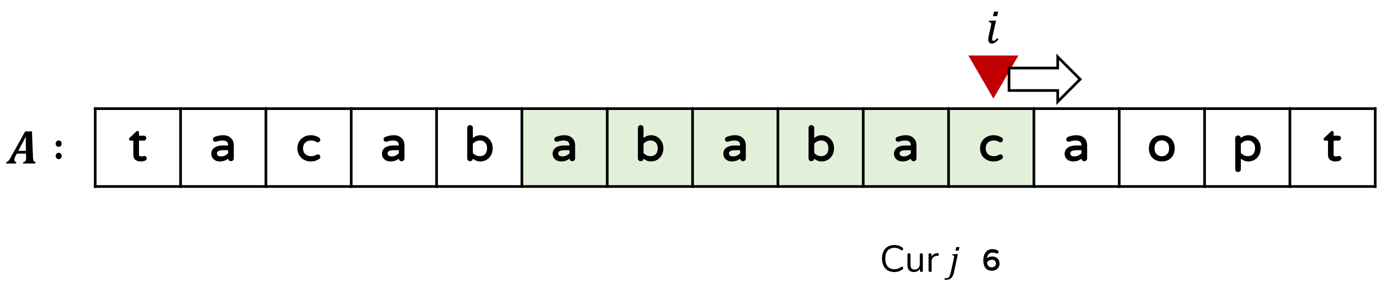


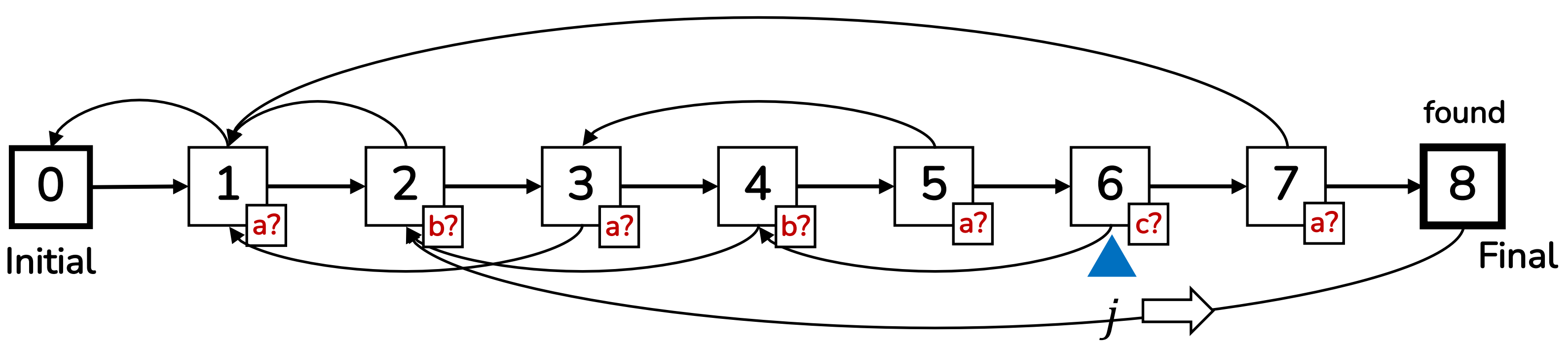
Search Phase with π (15)

P="ababaca"

❑ Compare $A[i]$ and $P[j]$

- ⇒ Match! Move i and j to the next



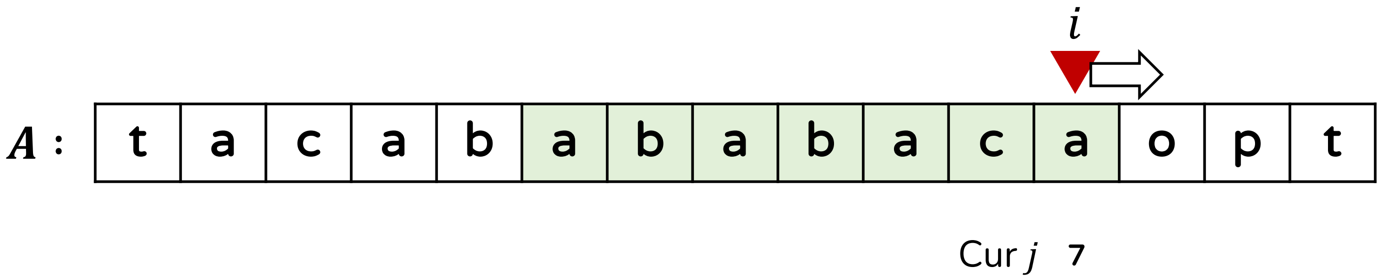


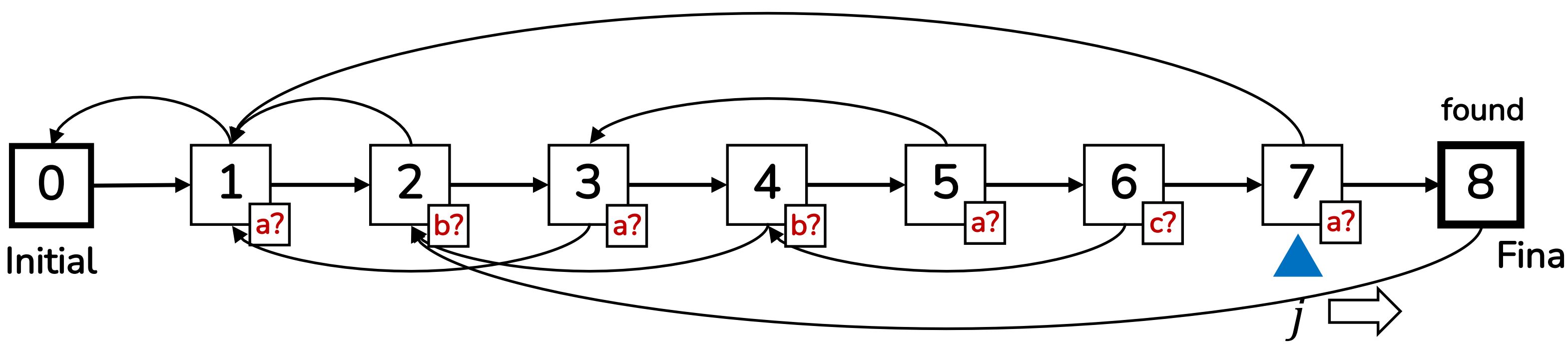
Search Phase with π (16)

P="ababaca"

❑ Compare $A[i]$ and $P[j]$

- ⇒ Match! Move i and j to the next

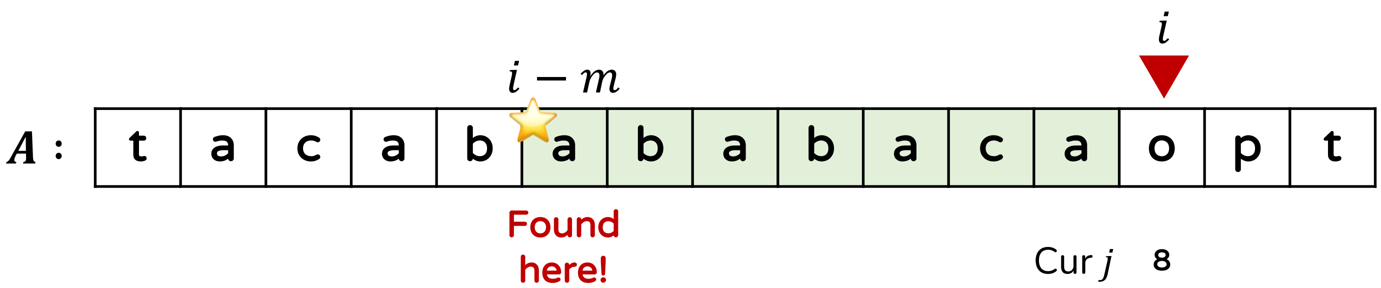


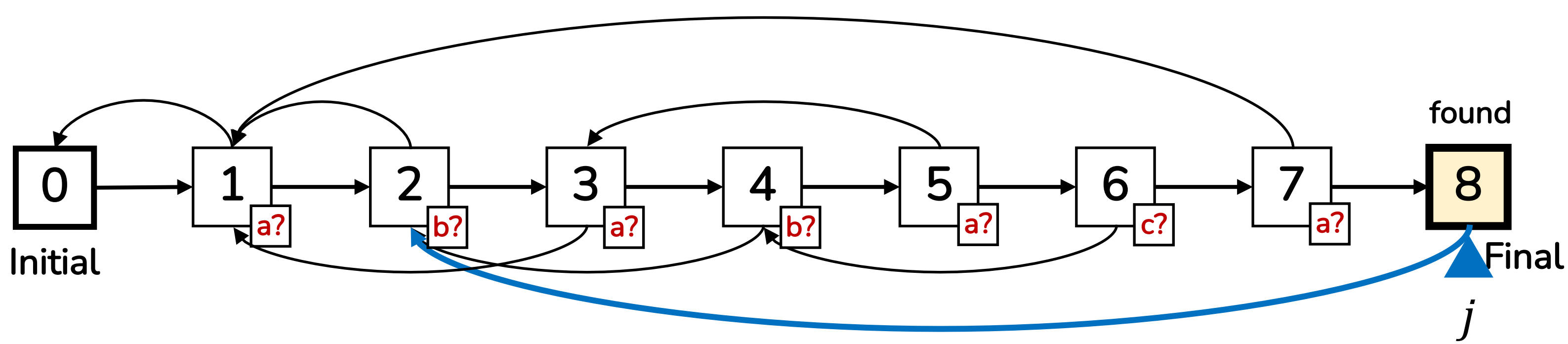


Search Phase with π (16)

P="ababaca"

- This is the final case ($j = m + 1$)
  - ⇒ Output "P is matched at $A[i - m]$", and go back $j \leftarrow \pi[j]$



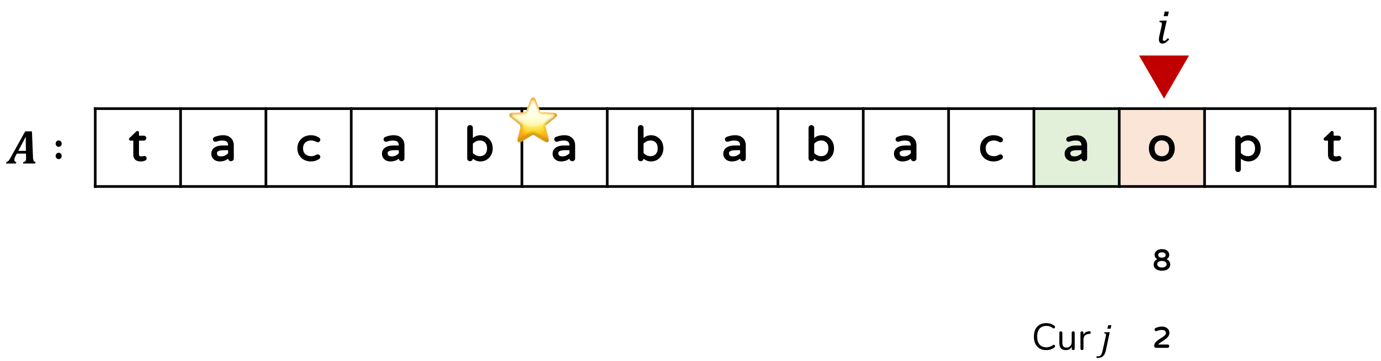


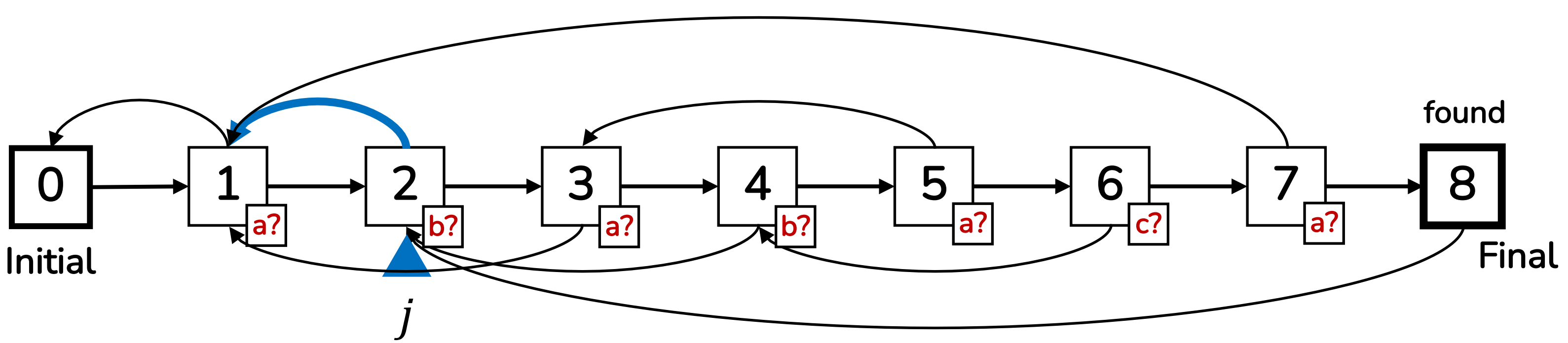
Search Phase with π (17)

P="ababaca"

❑ Compare $A[i]$ and $P[j]$

- ⇒ Failure! Move j back ($j \leftarrow \pi[j]$)





Search Phase with π (18)

P="ababaca"

❑ Compare $A[i]$ and $P[j]$

- ⇒ Failure! Move j back ($j \leftarrow \pi[j]$)
  - Repeat these until all characters in A are checked

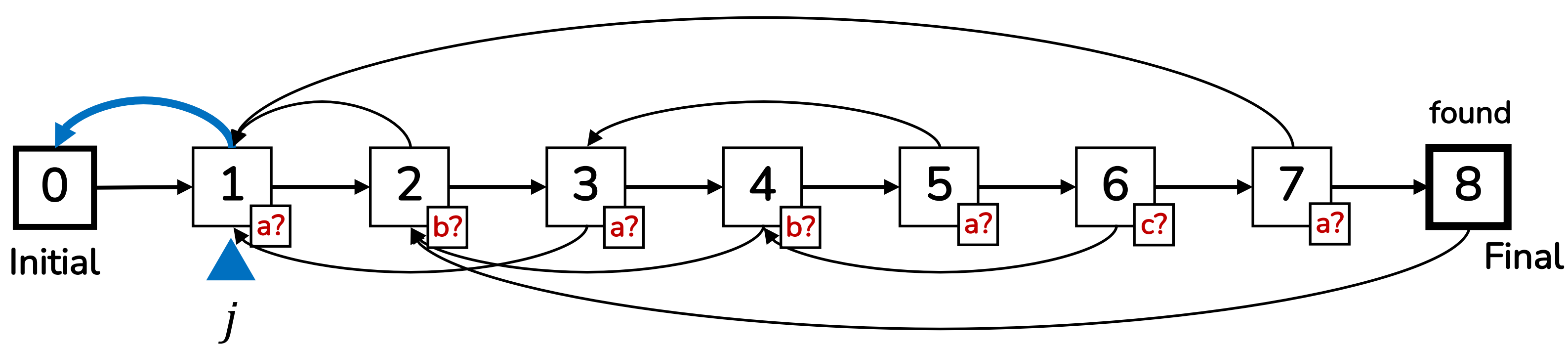
i

A :															
	t	a	c	a	b	a	b	a	b	a	c	a	o	p	t

8

2

Cur j 1



KMP Search Phase

❑ Pseudocode of the search phase

```
def KMP-search(A, P, π):
    # n: length of A (document string)
    # m: length of P (pattern string)

    i ← 1      # pointing to A
    j ← 1      # pointing to P

    while i <= n:
        if j == 0 or A[i] == P[j]:
            i ← i + 1
            j ← j + 1
        else:
            j ← π[j]

        if j == m+1:
            output "there is a matching at A[i-m]"
            j ← π[j]
```

Initial case (j=0)
Match case

Failure case

Final case

Time Complexity of Searching (1)

❑ It depends on # of iterations of the while loop

- **Match**: both i and j increase by 1
- **Failure & Final**: i does not change & j decreases to $\pi[j]$

```
def KMP-search(A, P, π):
    i ← 1  &  j ← 1
    while i <= n:
        if j == 0 or A[i] == P[j]:
            i ← i + 1                   Initial case (j=0)
            j ← j + 1                   Match case
        else:
            j ← π[j]                    Failure case

        if j == m+1:                    Final case
            output "there is a matching at A[i-m]"
            j ← π[j]
```

Time Complexity of Searching (2)

❑ Let's introduce a new variable $i + (i - j)$ as a trick

- For each iteration, $i + (i - j)$ increases by at least 1
  - **Match**: both i and j increase by 1
    - $\Rightarrow$ After then, $i + (i - j)$ increases by 1
  - **Failure**: i does not change & j decreases to $\pi[j]$
    - $\Rightarrow$ After then, $i + (i - j)$ increases by at least 1
- Note that $i + (i - j) \leq 2i$ because j cannot be negative, and
- $i + (i - j) \leq 2i \leq 2n$ because $i \leq n$ of the while-loop cond.
  - This implies that at the first, $i + (i - j)$ starts with 1 and increases by at least 1, but cannot exceed $2n$
- Therefore, the time complexity of searching is $O(n)$.

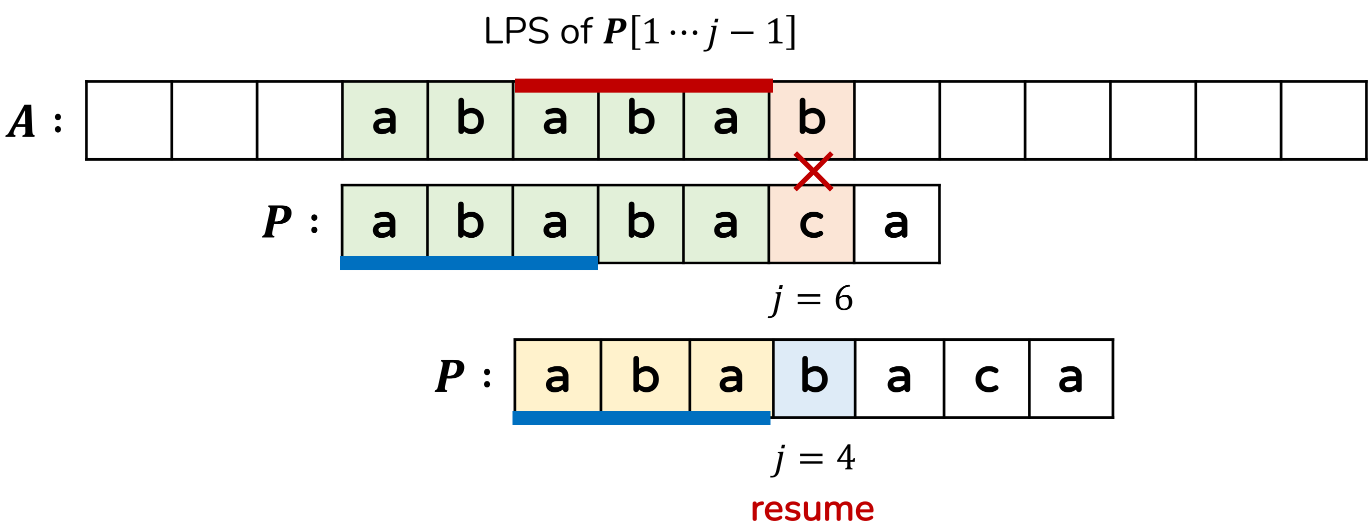
Outline

- ❑ Intuition of KMP algorithm
- ❑ Search phase
- ❑ Failure array construction phase

How To Construct π (1)

❑ Remind the meaning of $\pi[j]$

- $\boldsymbol{\pi}[j]$ indicates a resuming location in $\boldsymbol{P}$ when a match fails
  - The location is the next to the LPS of $\boldsymbol{P}[1 \cdots j-1]$
  - Thus, $\boldsymbol{\pi}[j] = 1 +$ length of LPS of $\boldsymbol{P}[1 \cdots j-1]$
- e.g., $\pi[6] = 1 + 3$ where the LPS is “aba” whose length is 3



How To Construct π (2)

❑ Naïve approach

- For each $\boldsymbol{P}[1 \cdots j-1]$, check if each of its proper prefixes is matched with its suffix, and pick the longest prefix
  - $\boldsymbol{\pi}[j]$ = 1 + length of LPS of $\boldsymbol{P}[1 \cdots j-1]$
- Repeat the above for $2 \leq j \leq m+1$

```
def naïve-KMP-failure-array(P):
    π[1] ← 0

    for j ← 2 to m+1:
        for len ← 1 to j-1:
            p ← get-prefix(P[1⋯j-1], len)
            s ← get-suffix(P[1⋯j-1], len)
            if p == s:
                π[j] ← 1 + len
```

$\boldsymbol{P}$:

a	b	a	b	a	c	a

$\boldsymbol{\pi} =$

0	1	2	3	4	5	6	7	8
×	0	1	1	2	3	4	1	2

How To Construct $\boldsymbol{\pi}$ (3)

❑ Time complexity of the naïve approach

- It takes $O(m^3)$ time
  - For each iteration, it takes $O(m^2)$ time at most to find the LPS
  - It repeats $O(m)$ times; thus, it is $O(m^3)$ in total

❑ Can we do this better?

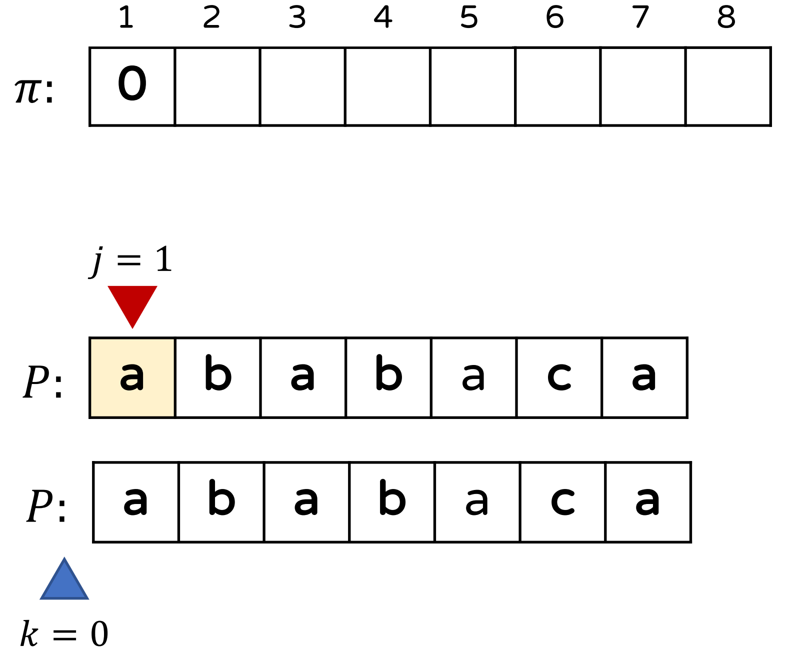
- As KMP's search phase, we can construct $\boldsymbol{\pi}$ in linear time
  - Main idea is to use previous information on LPS to build the current LPS
  - Surprisingly, it's similar to the search phase with $\boldsymbol{A} \leftarrow \boldsymbol{P}$
    - Because matching is equivalent to extending the prefix of $\boldsymbol{P}$ over $\boldsymbol{A}$
  - Derivation and correctness are out-of-scope. Refer to CLRS for proof

Fast Construction of π (1)

❑ Step 0 – Initialization

```
def KMP-failure-array(P):
    j ← 1 and k ← 0
    π[1] ← 0

    while j <= m:
        if k == 0 or P[j] == P[k]:
            j ← j + 1
            k ← k + 1
            π[j] ← k
        else:
            k ← π[k]
    return π
```

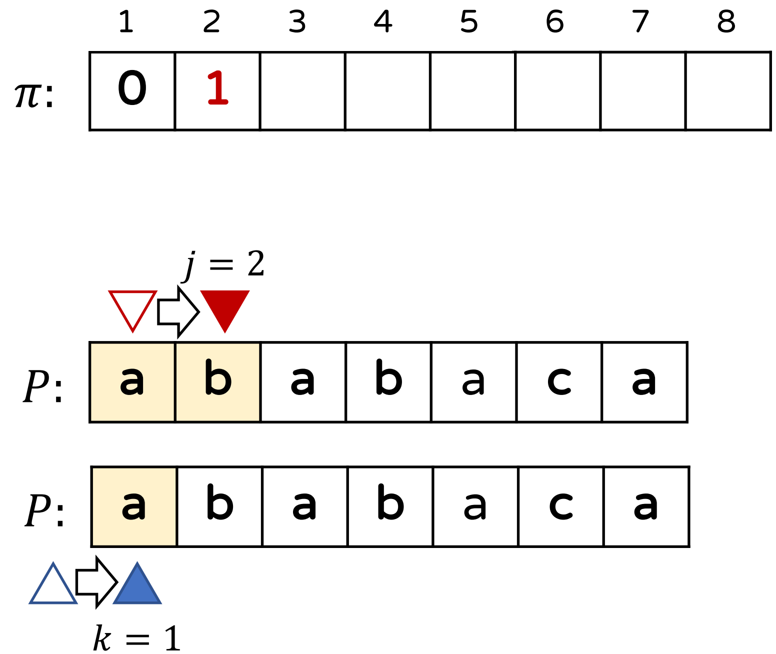


Fast Construction of π (2)

❑ Step 1 – Initial case

```
def KMP-failure-array(P):
    j ← 1 and k ← 0
    π[1] ← 0

    while j <= m:
        if k == 0 or P[j] == P[k]:
            j ← j + 1
            k ← k + 1
            π[j] ← k
        else:
            k ← π[k]
    return π
```



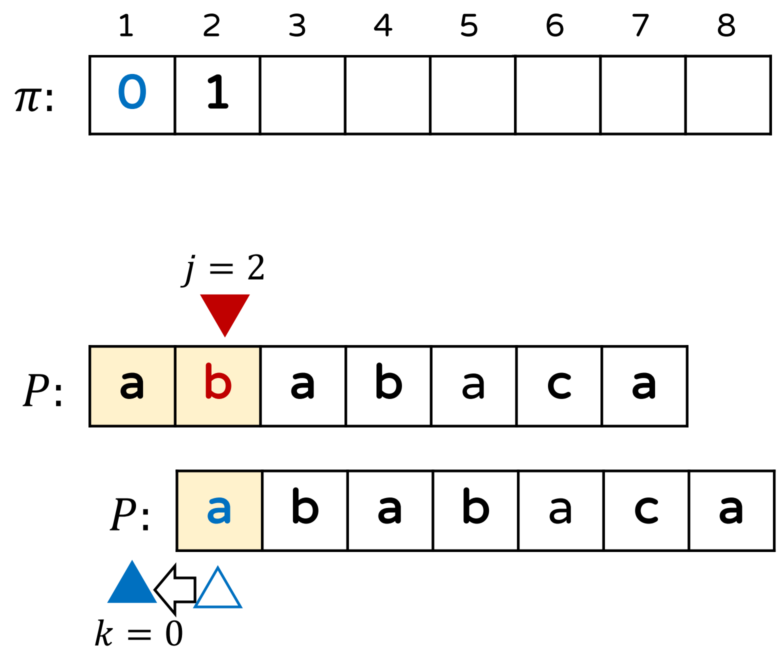
- k keeps tracking the position next to LPS of $P[1 \cdots j-1]$
- In this case, there is no LPS (or “”); thus, the position should be 1

Fast Construction of π (3)

❑ Step 2 – Failure case

```
def KMP-failure-array(P):
    j ← 1 and k ← 0
    π[1] ← 0

    while j <= m:
        if k == 0 or P[j] == P[k]:
            j ← j + 1
            k ← k + 1
            π[j] ← k
        else:
            k ← π[k]
   return π
```



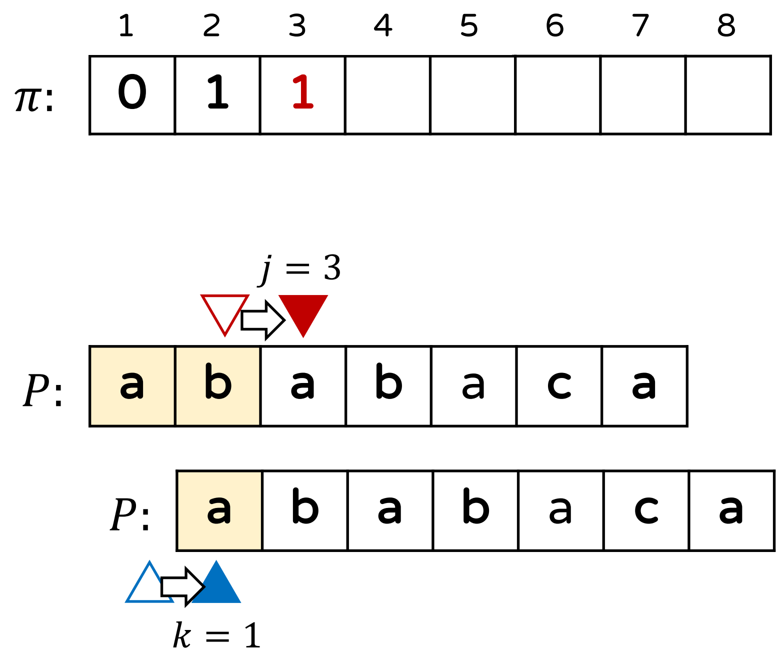
- Try to compare $P[j]$ and $P[k]$ to find next LPS of $P[1 \cdots j]$
- $P[j] \neq P[k]$; thus, go back to check other LPS (in this case, no more other LPS => initial)

Fast Construction of π (4)

❑ Step 3 – Initial case

```
def KMP-failure-array(P):
    j ← 1 and k ← 0
    π[1] ← 0

    while j <= m:
        if k == 0 or P[j] == P[k]:
            j ← j + 1
            k ← k + 1
            π[j] ← k
        else:
            k ← π[k]
    return π
```



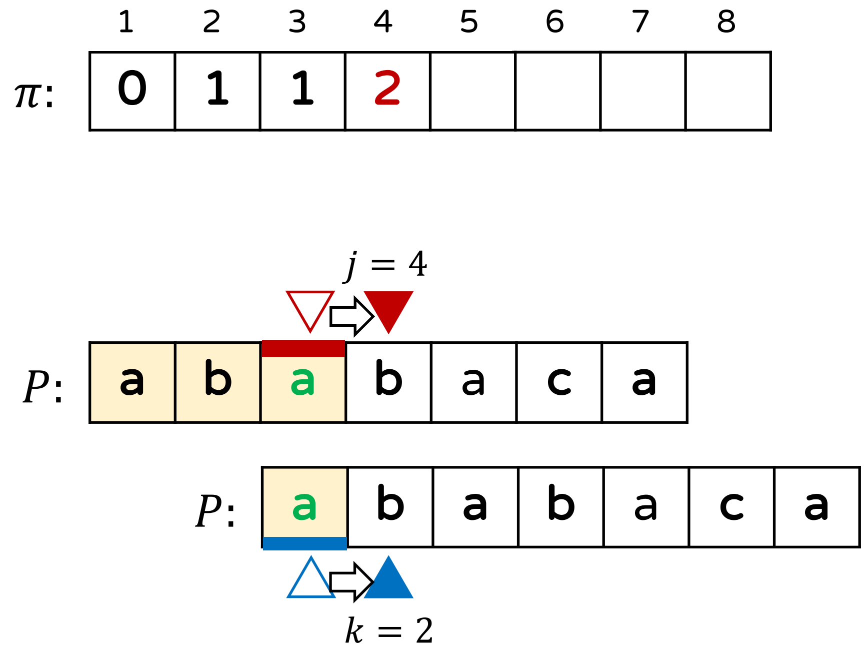
- k keeps tracking the position next to LPS of $P[1 \cdots j-1]$
- In this case, there is no LPS (or “”); thus, the position should be 1

Fast Construction of π (5)

❑ Step 4 – Match case

```
def KMP-failure-array(P):
    j ← 1 and k ← 0
    π[1] ← 0

    while j <= m:
        if k == 0 or P[j] == P[k]:
            j ← j + 1
            k ← k + 1
            π[j] ← k
        else:
            k ← π[k]
    return π
```



- k keeps tracking the position next to LPS of $P[1 \cdots j-1]$
- Here, LPS is "a", thus $\pi[4] = 1 + 1 = 2$

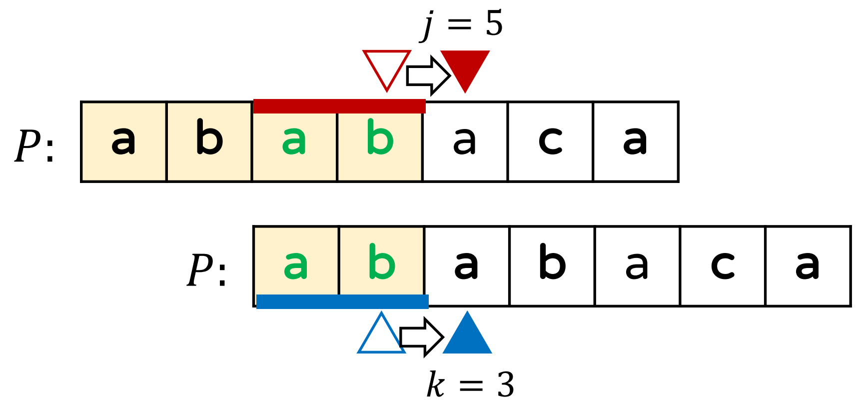
Fast Construction of π (6)

❑ Step 5 – Match case

```
def KMP-failure-array(P):
    j ← 1 and k ← 0
    π[1] ← 0

    while j <= m:
        if k == 0 or P[j] == P[k]:
            j ← j + 1
            k ← k + 1
            π[j] ← k
        else:
            k ← π[k]
    return π
```

	1	2	3	4	5	6	7	8
π:	0	1	1	2	3			



- k keeps tracking the position next to LPS of $P[1 \cdots j-1]$
- Here, LPS is “ab”, thus $\pi[5] = 2 + 1 = 3$
- Note that it used the previous LPS “a” to build the current LPS “ab” (**speed up**!)

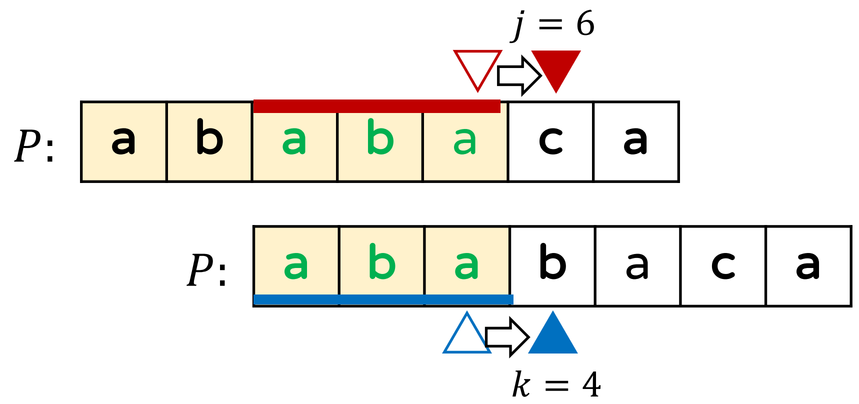
Fast Construction of π (7)

❑ Step 6 – Match case

```
def KMP-failure-array(P):
    j ← 1 and k ← 0
    π[1] ← 0

    while j <= m:
        if k == 0 or P[j] == P[k]:
            j ← j + 1
            k ← k + 1
            π[j] ← k
        else:
            k ← π[k]
  return π
```

	1	2	3	4	5	6	7	8
π:	0	1	1	2	3	4		



- k keeps tracking the position next to LPS of $P[1 \cdots j-1]$
- Here, LPS is “aba”, thus $\pi[6] = 3 + 1 = 4$
- Note that it used the previous LPS “ab” to build the current LPS “aba” (**speed up**!)

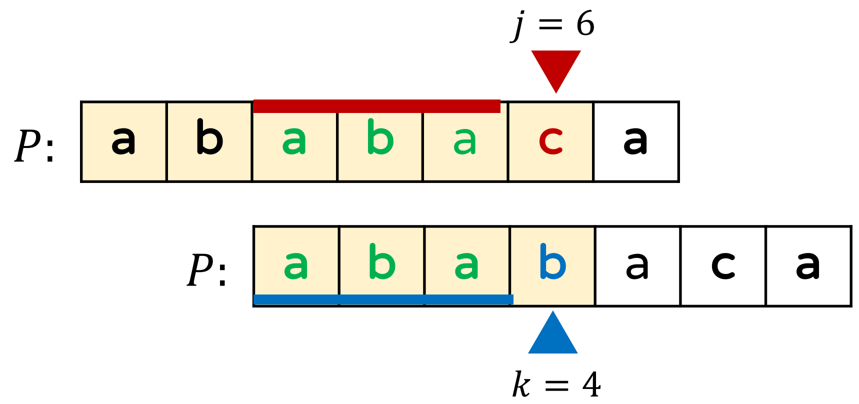
Fast Construction of π (8)

❑ Step 7-1 – Failure case

```
def KMP-failure-array(P):
    j ← 1 and k ← 0
    π[1] ← 0

    while j <= m:
        if k == 0 or P[j] == P[k]:
            j ← j + 1
            k ← k + 1
            π[j] ← k
        else:
            k ← π[k]
    return π
```

	1	2	3	4	5	6	7	8
π:	0	1	1	2	3	4		



- Try to compare $P[j]$ and $P[k]$ to find next LPS of $P[1 \cdots j]$
- But, $\boldsymbol{P}[j] \neq \boldsymbol{P}[k]$, meaning we cannot use the previous LPS “aba”
- This is equal to that a match fails at $k \Rightarrow$ move k back to $\pi[k]$

Fast Construction of π (9)

❑ Step 7-2 – Failure case

```
def KMP-failure-array(P):
    j ← 1 and k ← 0
    π[1] ← 0

    while j <= m:
        if k == 0 or P[j] == P[k]:
            j ← j + 1
            k ← k + 1
            π[j] ← k
        else:
            k ← π[k]
    return π
```

π:

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

$j = 6$

P:

a	b	a	b	a	c	a

P:

a	b	a	b	a	c	a

$k = 2$

- After k is moved, we have one more change to find a shorter LPS based on “a”
- To do that, compare $\boldsymbol{P}[j]$ and $\boldsymbol{P}[k]$

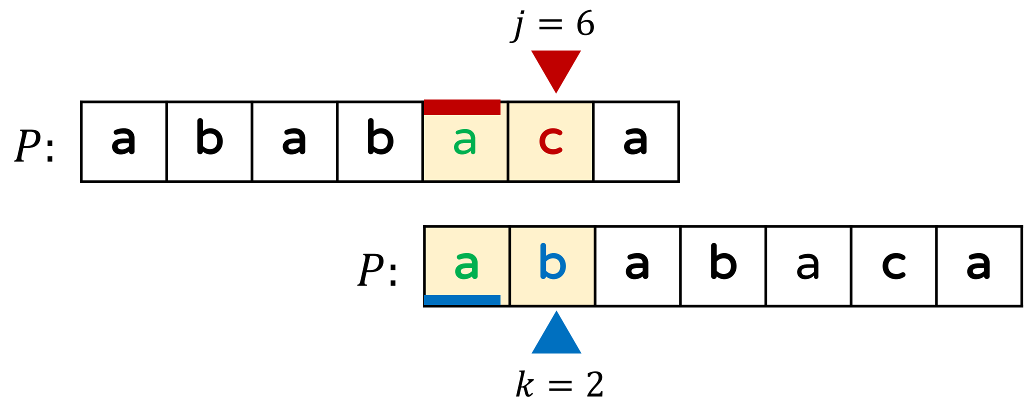
Fast Construction of π (10)

❑ Step 8-1 – Failure case

```
def KMP-failure-array(P):
    j ← 1 and k ← 0
    π[1] ← 0

    while j <= m:
        if k == 0 or P[j] == P[k]:
            j ← j + 1
            k ← k + 1
            π[j] ← k
        else:
            k ← π[k]
    return π
```

	1	2	3	4	5	6	7	8
π:	0	1	1	2	3	4		



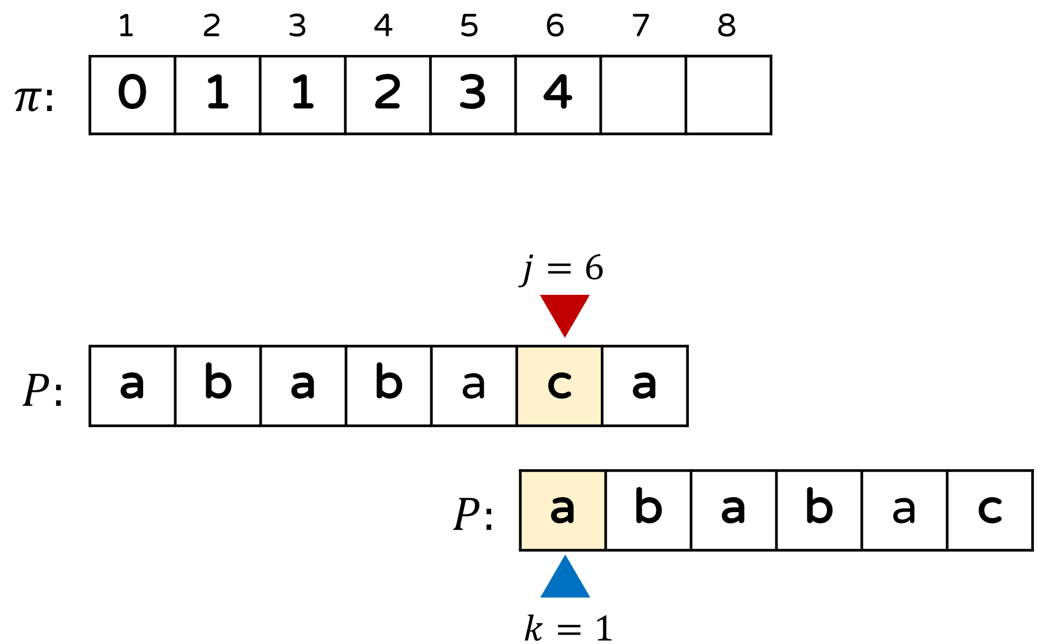
- This is equal to that a match fails at $k \Rightarrow$ move k back to $\pi[k]$

Fast Construction of π (11)

❑ Step 8-2 – Failure case

```
def KMP-failure-array(P):
    j ← 1 and k ← 0
    π[1] ← 0

    while j <= m:
        if k == 0 or P[j] == P[k]:
            j ← j + 1
            k ← k + 1
            π[j] ← k
        else:
            k ← π[k]
    return π
```



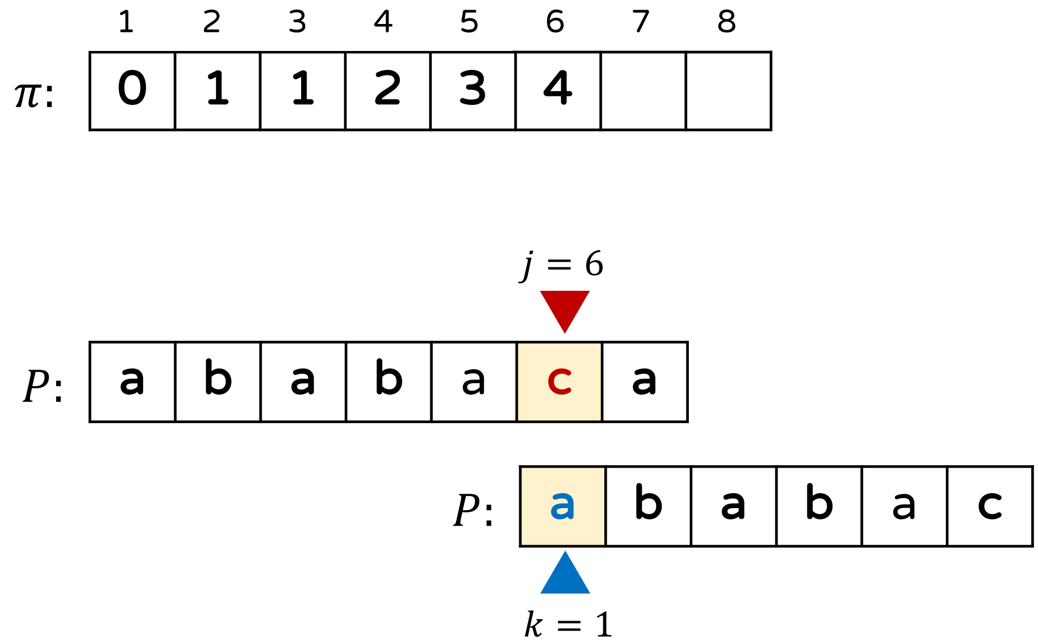
- After k is moved, no more LPS here

Fast Construction of π (12)

❑ Step 9-1 – Failure case

```
def KMP-failure-array(P):
    j ← 1 and k ← 0
    π[1] ← 0

    while j <= m:
        if k == 0 or P[j] == P[k]:
            j ← j + 1
            k ← k + 1
            π[j] ← k
        else:
            k ← π[k]
    return π
```

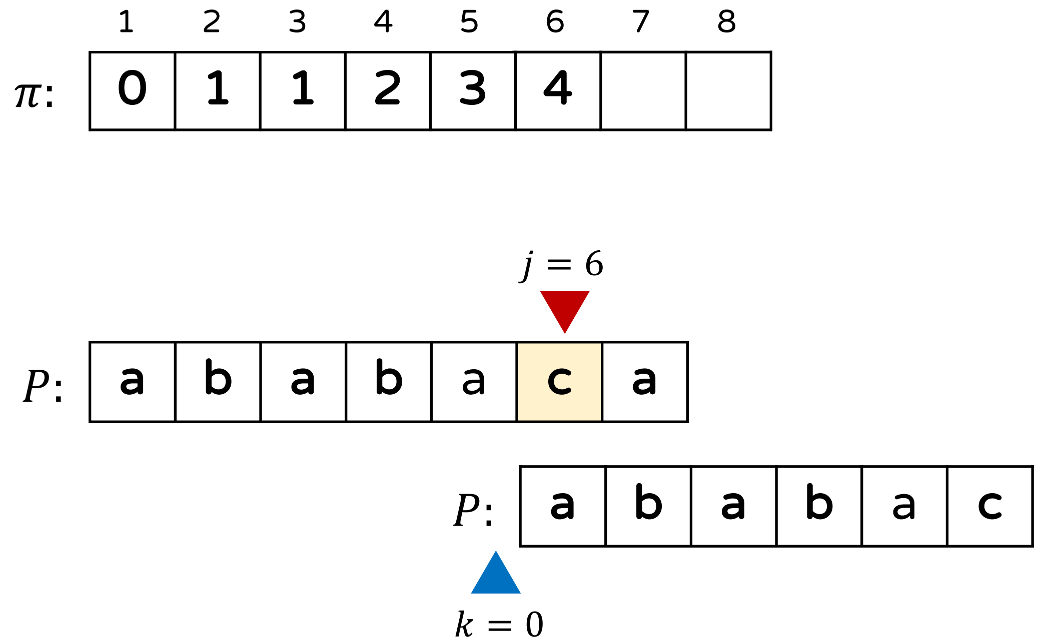


Fast Construction of π (13)

❑ Step 9-2 – Failure case

```
def KMP-failure-array(P):
    j ← 1 and k ← 0
    π[1] ← 0

    while j <= m:
        if k == 0 or P[j] == P[k]:
            j ← j + 1
            k ← k + 1
            π[j] ← k
        else:
            k ← π[k]
    return π
```

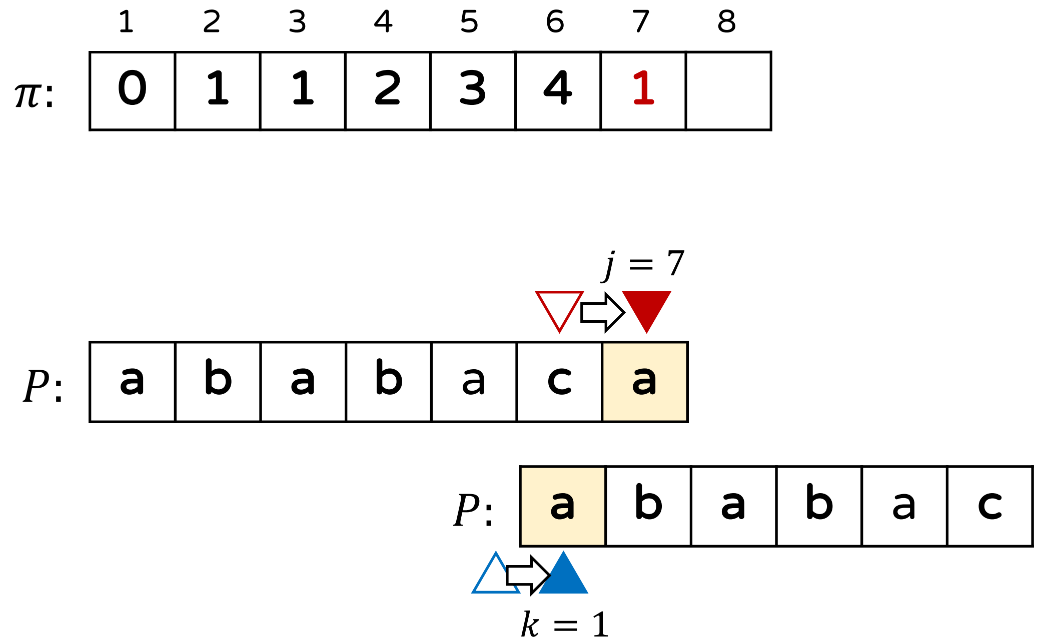


Fast Construction of π (14)

❑ Step 10 – Initial case

```
def KMP-failure-array(P):
    j ← 1 and k ← 0
    π[1] ← 0

    while j <= m:
        if k == 0 or P[j] == P[k]:
            j ← j + 1
            k ← k + 1
            π[j] ← k
        else:
            k ← π[k]
    return π
```

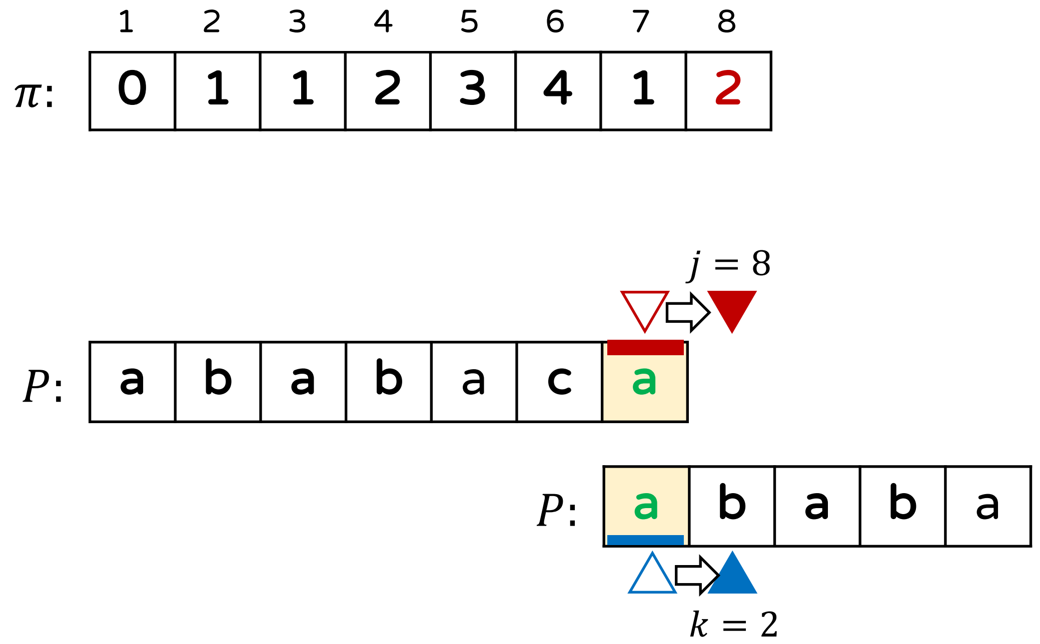


Fast Construction of π (15)

❑ Step 11 – Match case

```
def KMP-failure-array(P):
    j ← 1 and k ← 0
    π[1] ← 0

    while j <= m:
        if k == 0 or P[j] == P[k]:
            j ← j + 1
            k ← k + 1
            π[j] ← k
        else:
            k ← π[k]
  return π
```



Complexity Analysis of KMP

❑ Time complexity of the construction phase

- Similar to the search phase, it takes $O(m)$ time
  - By introducing a new variable $j + (j - k)$ as a trick

❑ Total complexity of KMP algorithm

- First, construct the failure array $\boldsymbol{\pi}$ from the pattern $\boldsymbol{P}$
  - $\boldsymbol{\pi} \leftarrow$KMP-failure-array($\boldsymbol{P}$) takes $O(m)$ time
- Second, match pattern $\boldsymbol{P}$ over document $\boldsymbol{A}$ with $\boldsymbol{\pi}$
  - KMP-search($\boldsymbol{A}$, $\boldsymbol{P}$, $\boldsymbol{\pi}$) takes $O(n)$ time
- In total, KMP algorithm takes $O(m + n)$ time
  - Faster than the automata algorithm taking $O(|\boldsymbol{\Sigma}|m^3 + n)$ time
- It uses $O(m)$ extra space for $\boldsymbol{\pi}$

What You Need To Know

❑ KMP algorithm

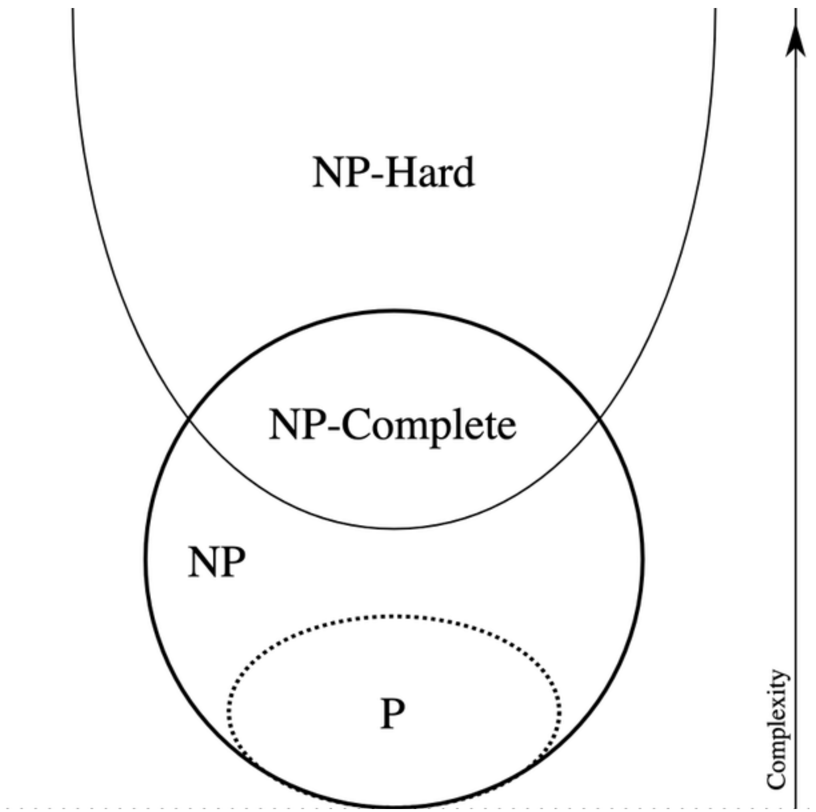
- Restart from a resuming location when a match fails, not from scratch
- The failure array generated from the pattern knows where we go back to for the failure

Algorithm	Time			Space	
	Preprocessing	Searching	Total	Input	Extra
Naïve	$O(1)$	$O(mn)$	$O(mn)$	$O(m+n)$	$O(1)$
Rabin-Karp	$O(m)$	$O(n+Fm)$	$O(n+Fm)$		$O(1)$
Automata	$O(\|\Sigma\|m^3)$	$O(n)$	$O(\|\Sigma\|m^3+n)$		$O(\|\Sigma\|m)$
KMP	$O(m)$	$O(n)$	$O(m+n)$		$O(m)$

* Rabin-karp's search phase shows $O(n)$ average-case time and $O(mn)$ worst-case time
* Automata can be constructed in $O(|\Sigma|m)$ time using the optimized version

In Next Lecture

- NP complexity theory



Thank You