# String Matching

Inge Li Gørtz

CLRS 32

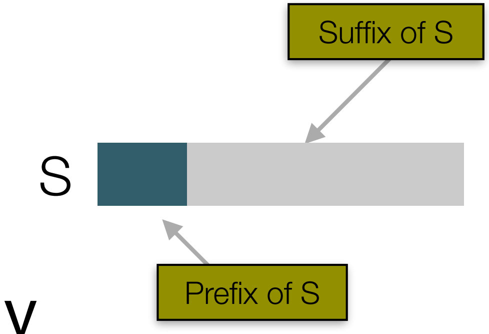
# String Matching

- String matching problem:
  - string T (text) and string P (pattern) over an alphabet Σ.
  - |T| = n, |P| = m.
  - Report all starting positions of occurrences of P in T.

P = a b a b a c a

T = b a c b a b a b a b a b a c a b

# Strings

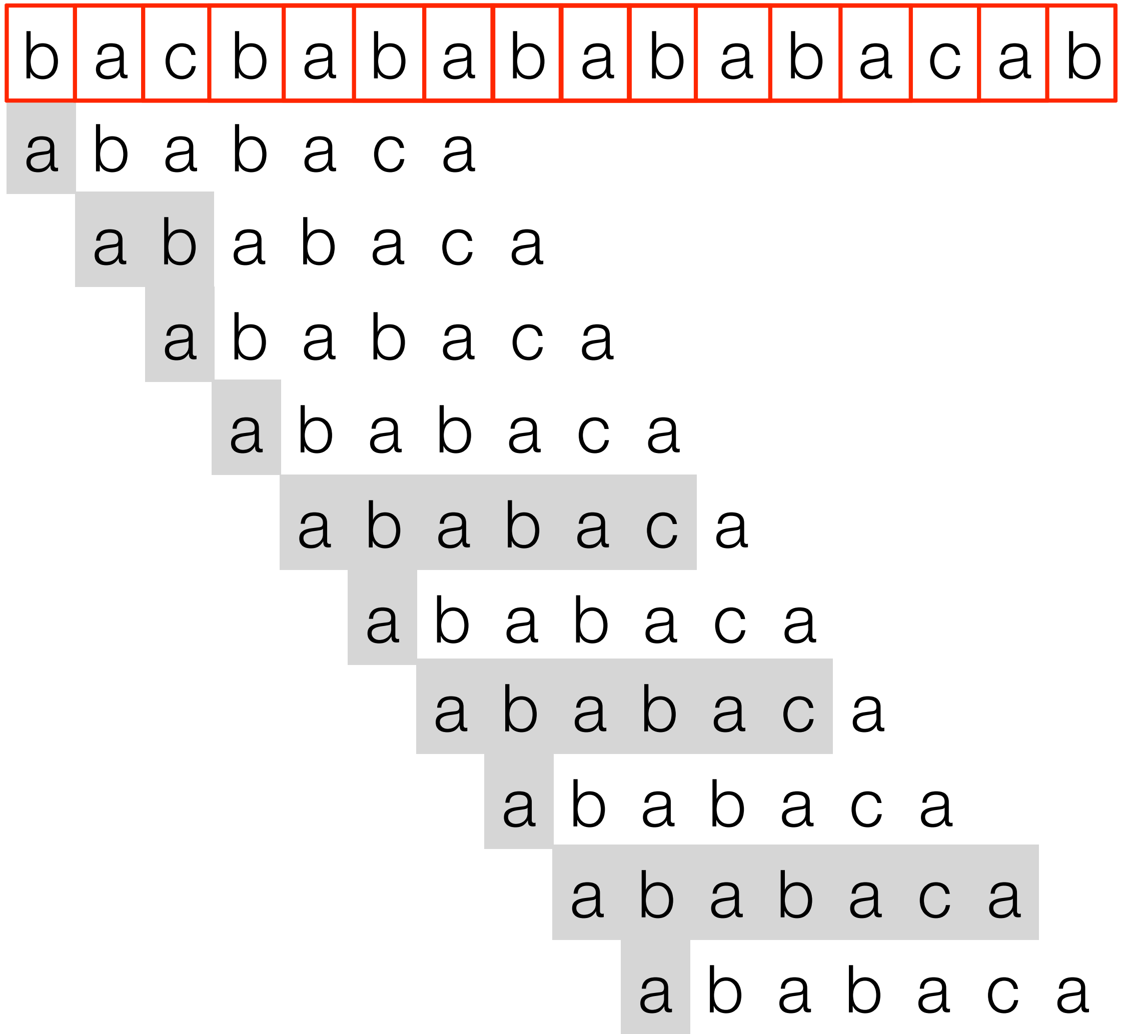



- ε: empty string
- prefix/suffix: v=xy:
  - x *prefix* of v, if y ≠ ε x is a *proper prefix* of v
  - y *suffix* of v, if y ≠ ε x is a *proper suffix* of v.
- Example: S = aabca
  - The suffixes of S are: aabca, abca, bca, ca and a.
  - The strings abca, bca, ca and a are proper suffixes of S.

# String Matching

- Knuth-Morris-Pratt (KMP)
- Finite automaton

# A naive string matching algorithm



# Improving the naive algorithm

P = a a a b a b a

T = a a a b a a [redacted]

a a a b a b a

a a a b a b a

# Exploiting what we know from pattern

P = a b a b a c a

T = a b a b a a x

a b a b a c a How much should we shift the pattern? 5

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

T = a b a b a b x

a b a b a c a How much should we shift the pattern? 2

a b a b a c a

T = a b a b a c x

a b a b a c a How much should we shift the pattern? 0

a b a b a c a

# Exploiting what we know from pattern

P = a b a b a c a

T = a b a b a a x

a b a b a c a Which character in the pattern should we compare to x? 2

a b a b a c a

T = a b a b a b x

a b a b a c a Which character in the pattern should we compare to x? 5

a b a b a c a

T = a b a b a c x

a b a b a c a Which character in the pattern should we compare to x? 7

a b a b a c a

# Exploiting what we know from pattern

P = a b a b a c a

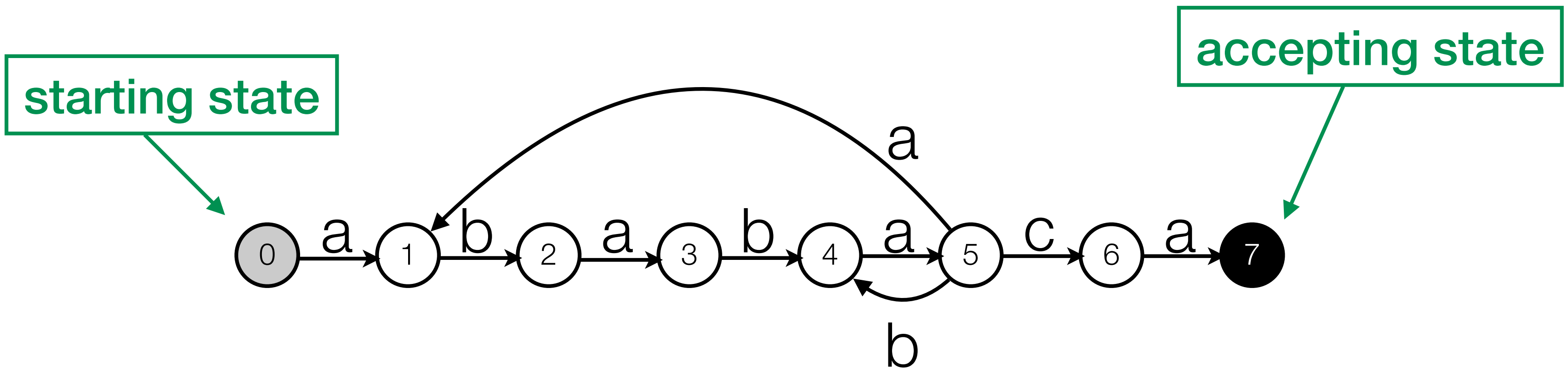
```
T = a b a b a a x
    a b a b a c a        How much can we "reuse"?   1
              a b a b a c a
```

```
T = a b a b a b x
    a b a b a c a        How much can we "reuse"?   4
        a b a b a c a
```

```
T = a b a b a c x
    a b a b a c a        How much can we "reuse"?   6
    a b a b a c a
```

# Finite Automaton

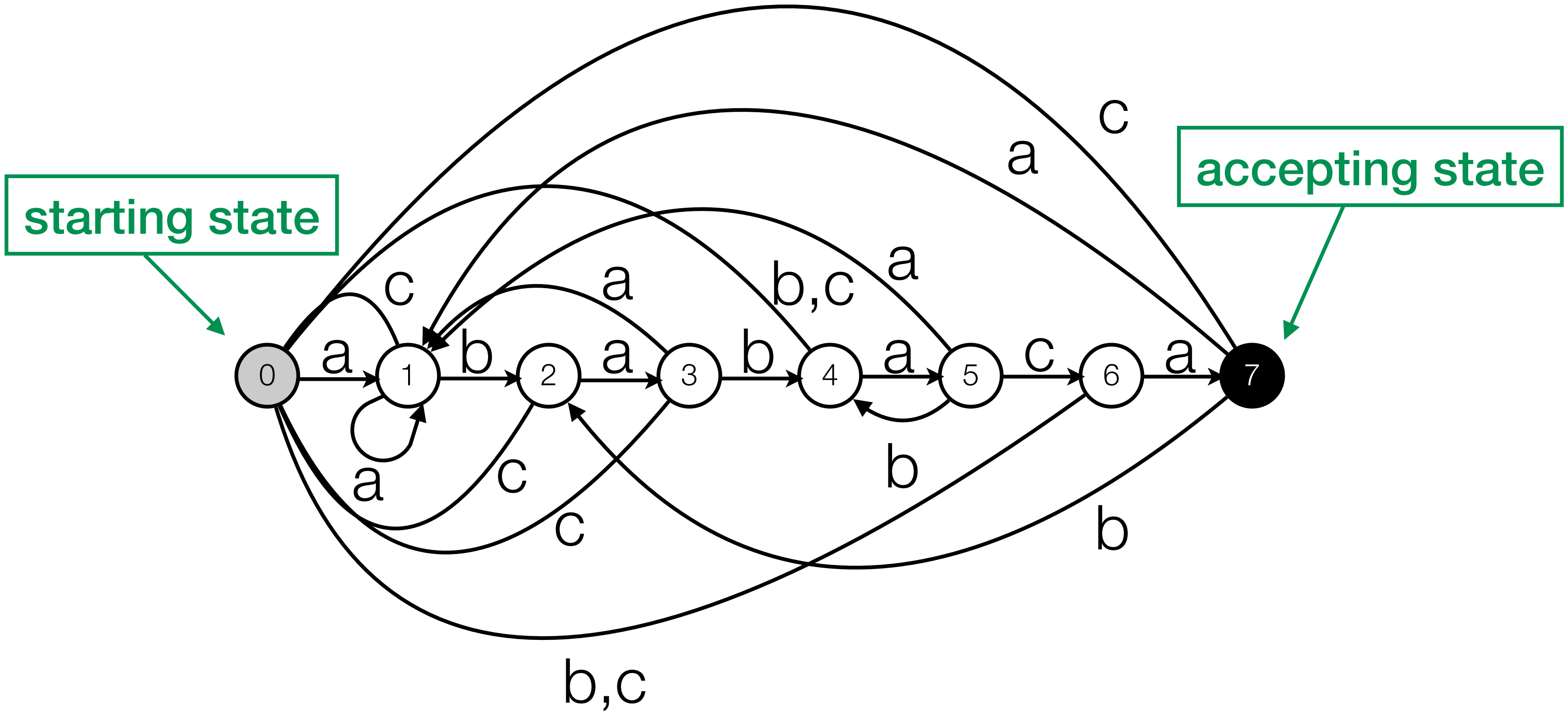
- Finite automaton: alphabet Σ = {a,b,c}. P = ababaca.



# Finite Automaton

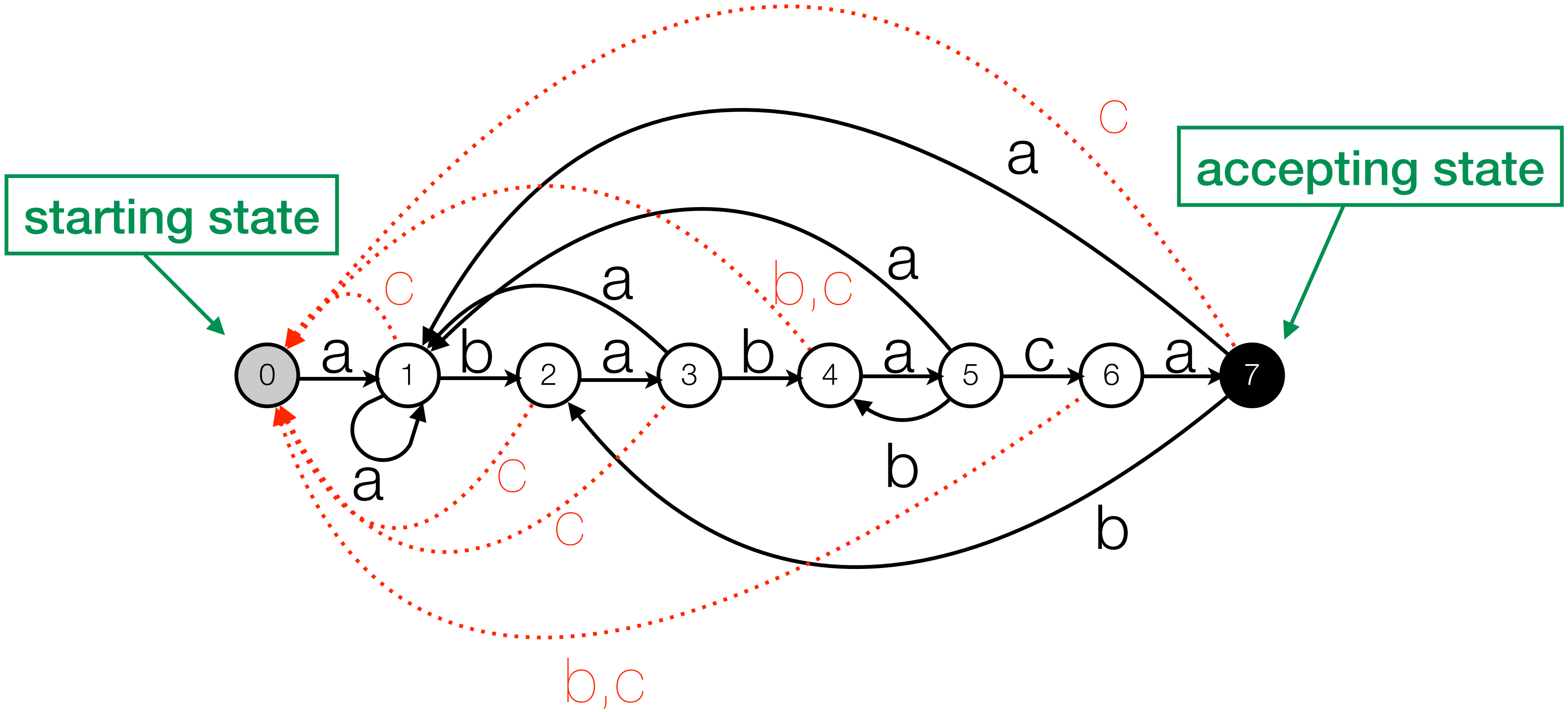
# Finite Automaton

- Finite automaton: alphabet $\Sigma = \{a,b,c\}$. P = ababaca.



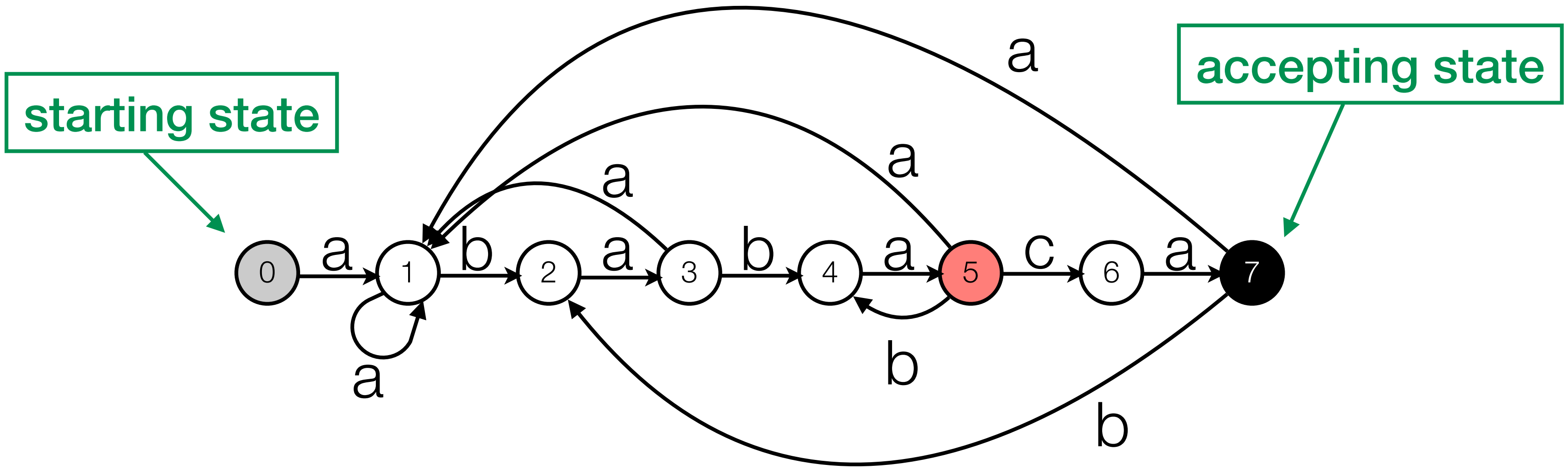
# Finite Automaton

- Finite automaton: alphabet Σ = {a,b,c}. P = ababaca.



# Finite Automaton

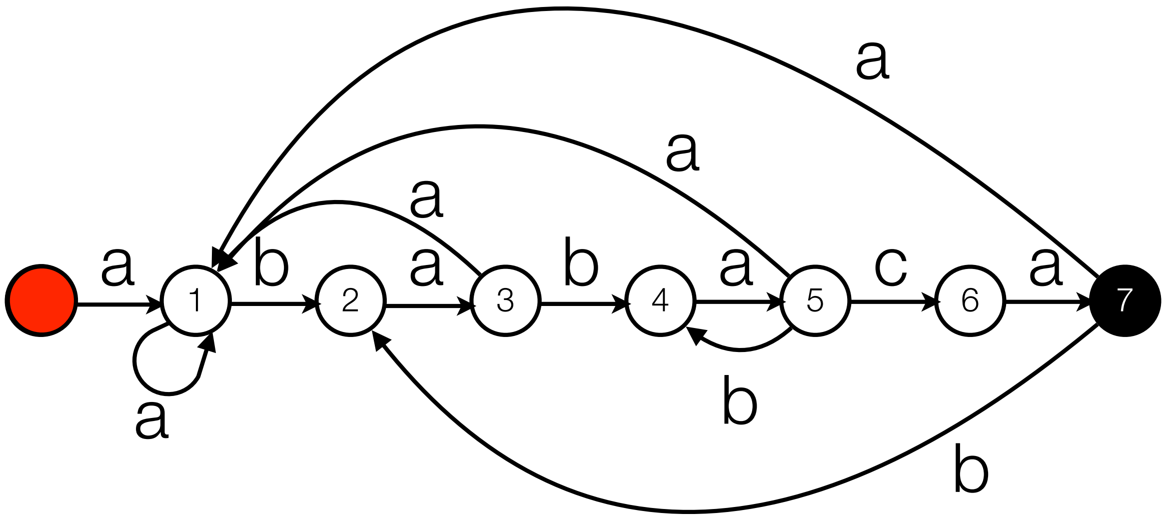
- Finite automaton: alphabet Σ = {a,b,c}. P = ababaca.




- State j: arc with character $\alpha$ goes to state i ≤ j +1 such that $P[1..i]$ is the longest prefix of $P$ that is a suffix of $P[1 \ldots j] \cdot \alpha$.

# Finite Automaton

- Finite automaton: alphabet $\Sigma = \{a,b,c\}$. P = ababaca.

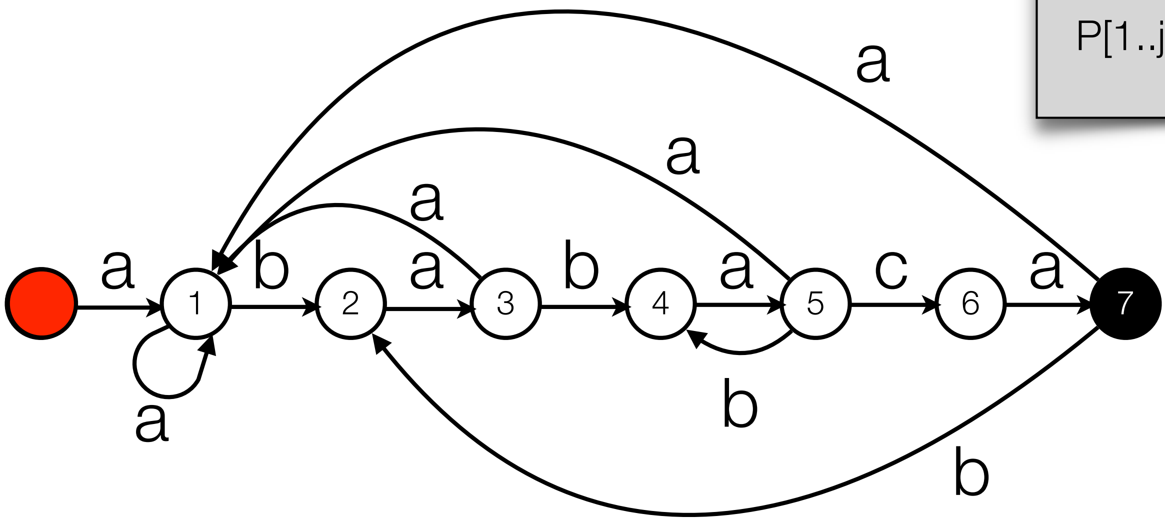



T = b a c b a b a b a b a b a c a b

# Finite Automaton

- Finite automaton: alphabet $\Sigma$ = {a,b,c}. P = ababaca.

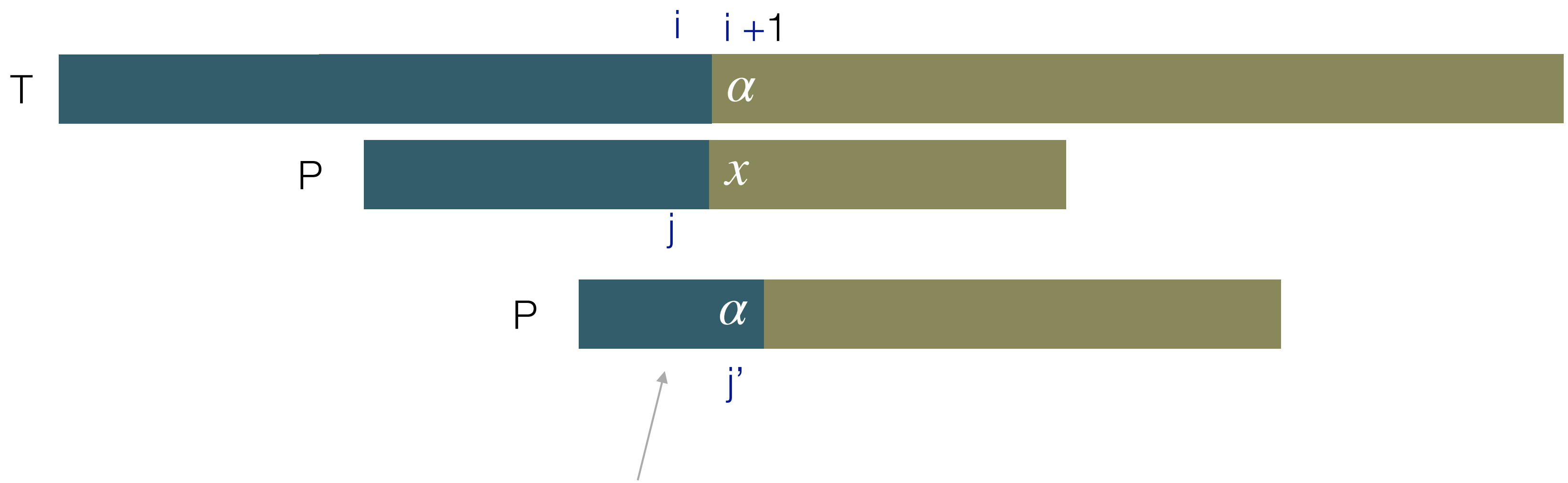
If we are in state j after reading T[1..i], then
P[1..j] is the longest prefix of P that is a suffix of T[1..i].




T = $\boxed{\text{b}}$ a c b a b a b a b a b a c a b

# Finite Automaton

If we are in state j after reading T[1..i], then
P[1..j] is the longest prefix of P that is a suffix of T[1..i].




P[1..j’] longest prefix of P that is a suffix of T[1..i+1].

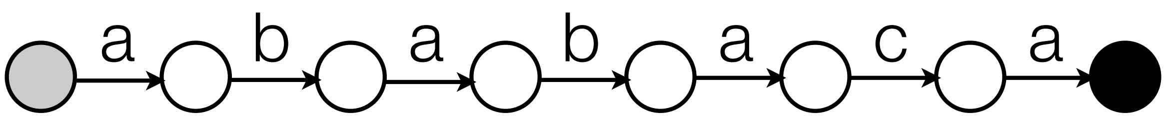
P[1..j’-1] is a prefix of P[1..j].

$\Rightarrow$

P[1..j’] longest prefix of P that is a suffix of P[1..j] $\cdot$ $\alpha$.

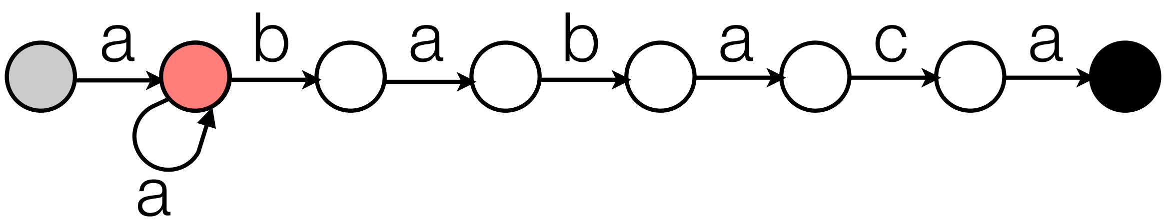
# Finite Automaton Construction

- Finite automaton: alphabet $\Sigma$ = {a,b,c}. P= ababaca.



# Finite Automaton Construction

- Finite automaton: alphabet $\Sigma = \{a,b,c\}$. P= ababaca.




**read ‘a’?**

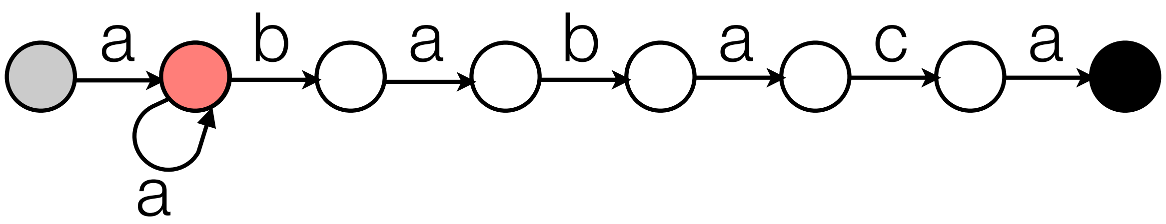
**longest prefix of P that is a proper suffix of ‘aa’ = ‘a’**

Matched until now: a a

P: a b a b a c a

# Finite Automaton Construction

- Finite automaton: alphabet Σ = {a,b,c}. P= ababaca.



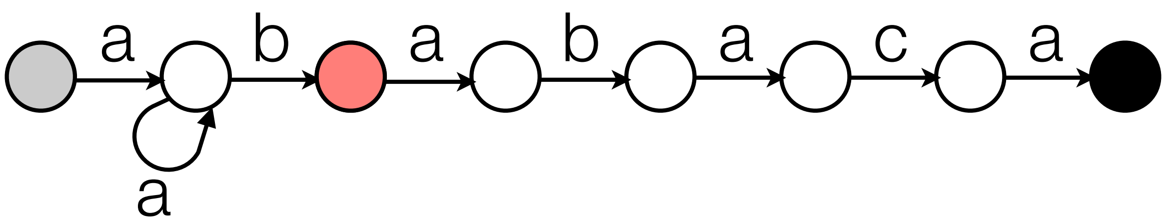

**read ‘c’?**

**longest prefix of P that is a proper suffix of ‘ac’ = ‘ ’**

| Matched until now: | a | c | | | | | |
|---|---|---|---|---|---|---|---|
| P: | a | b | a | b | a | c | a |

# Finite Automaton Construction

- Finite automaton: alphabet $\Sigma$ = {a,b,c}. P= ababaca.




**read ‘b’?**

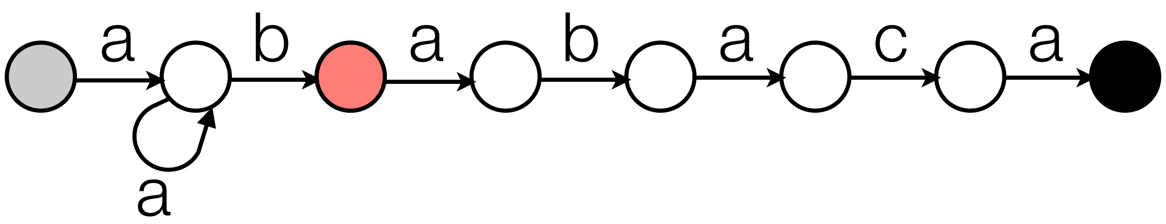
**longest prefix of P that is a proper suffix of ‘abb’ = ‘ ’**

Matched until now: a b b

P: a b a b a c a

# Finite Automaton Construction

- Finite automaton: alphabet Σ = {a,b,c}. P= ababaca.



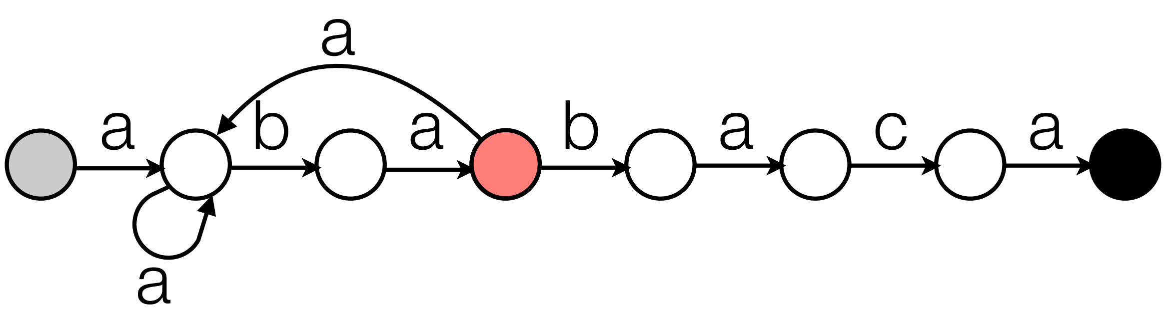

**read ‘c’?** **longest prefix of P that is a proper suffix of ‘abc’ = ‘ ’**

Matched until now: a b c

P: a b a b a c a

# Finite Automaton Construction

- Finite automaton: alphabet $\Sigma = \{a,b,c\}$. P= ababaca.



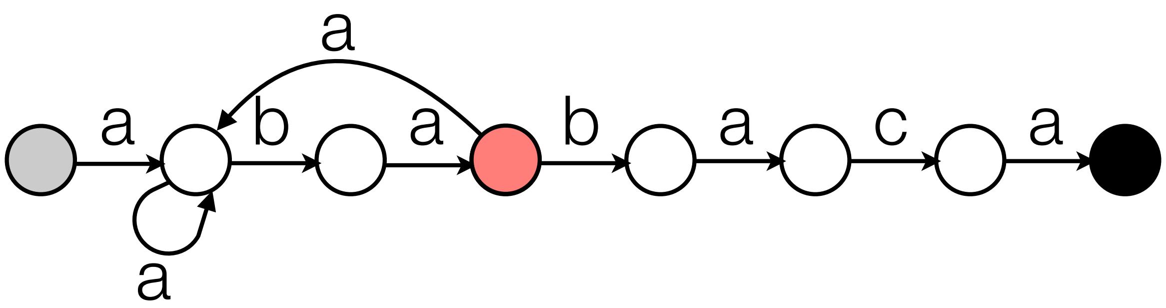

**read ‘a’?** **longest prefix of P that is a proper suffix of ‘abaa’ = ‘a’**

Matched until now: a b a a

P: a b a b a c a

# Finite Automaton Construction

- Finite automaton: alphabet $\Sigma$ = {a,b,c}. P= ababaca.



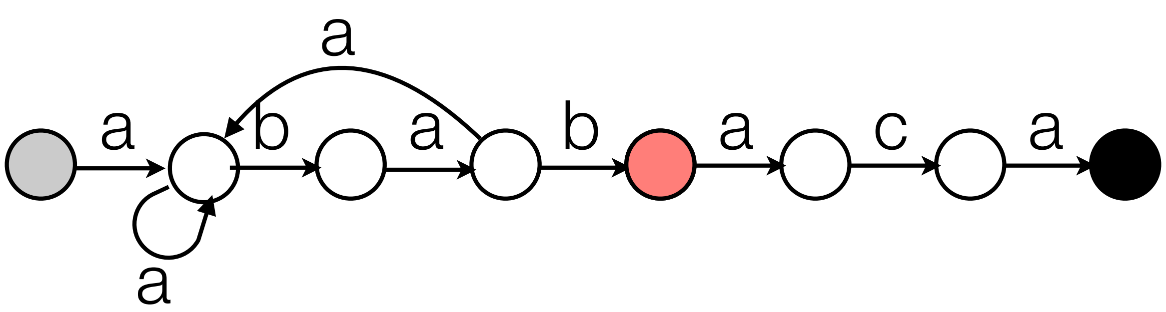

**read ‘c’?** **longest prefix of P that is a proper suffix of ‘abac’ = ‘ ’**

Matched until now: a b a c

P: a b a b a c a

# Finite Automaton Construction

- Finite automaton: alphabet Σ = {a,b,c}. P= ababaca.




**read ‘b’?**

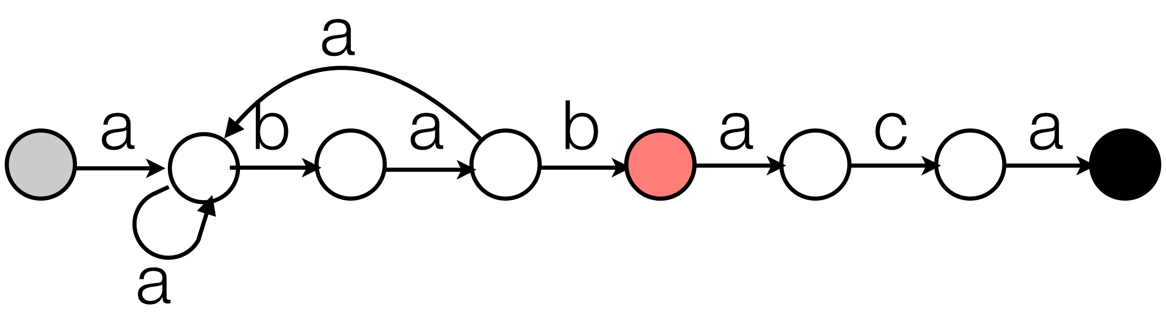
**longest prefix of P that is a proper suffix of ‘ababb’ = ‘ ’**

Matched until now: a b a b b

P: a b a b a c a

# Finite Automaton Construction

- Finite automaton: alphabet Σ = {a,b,c}. P= abababaca.

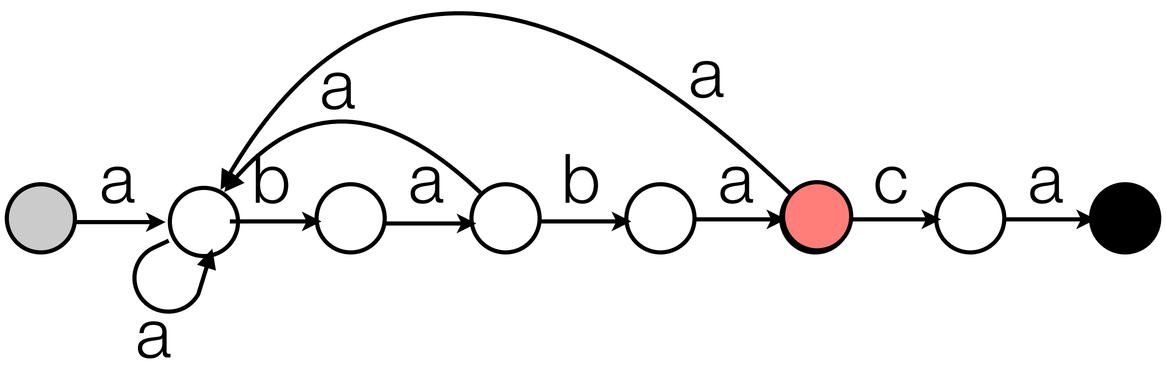



**read 'c'?** **longest prefix of P that is a proper suffix of 'ababc' = ' '**

| | |
|---|---|
| Matched until now: | a b a b c |
| P: | a b a b a c a |

# Finite Automaton Construction

- Finite automaton: alphabet Σ = {a,b,c}. P= ababaca.



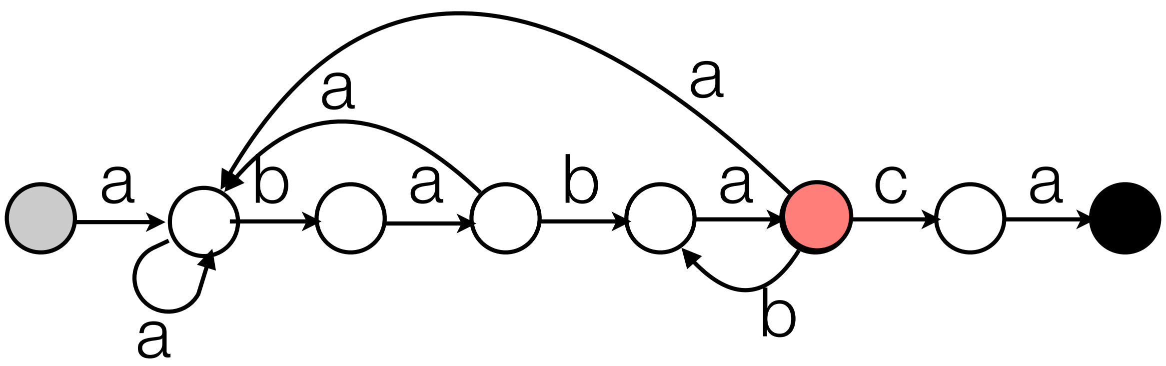

**read ‘a’?** **longest prefix of P that is a proper suffix of ‘ababaa’ = ‘a’**

Matched until now: a b a b a a

P: a b a b a c a

# Finite Automaton Construction

- Finite automaton: alphabet $\Sigma$ = {a,b,c}. P= ababaca.



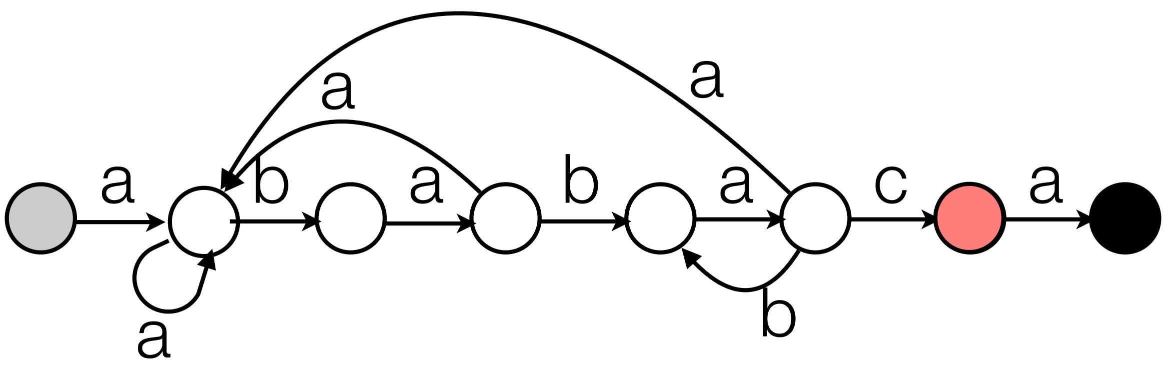

read ‘b’?

longest prefix of P that is a proper suffix of ‘ababaa’ = ‘abab’

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Matched until now: | a | b | a | b | a | b | |
| P: | a | b | a | b | a | c | a |

# Finite Automaton Construction

- Finite automaton: alphabet Σ = {a,b,c}. P= ababaca.

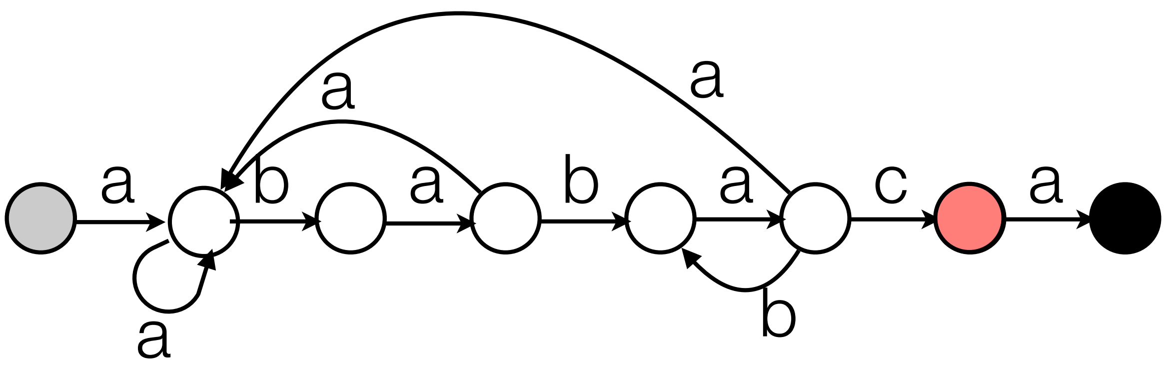



**read ‘b’?**

**longest prefix of P that is a proper suffix of ‘ababacb’ = ‘ ’**

# Finite Automaton Construction

- Finite automaton: alphabet Σ = {a,b,c}. P= ababaca.

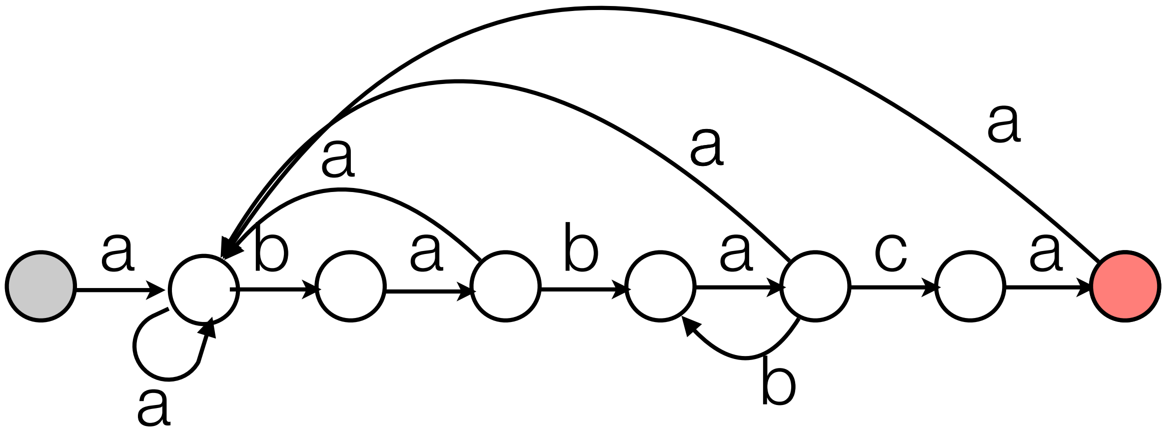



read ‘c’?

longest prefix of P that is a proper suffix of ‘ababacc’ = ‘ ’

# Finite Automaton Construction

- Finite automaton: alphabet Σ = {a,b,c}. P= ababaca.

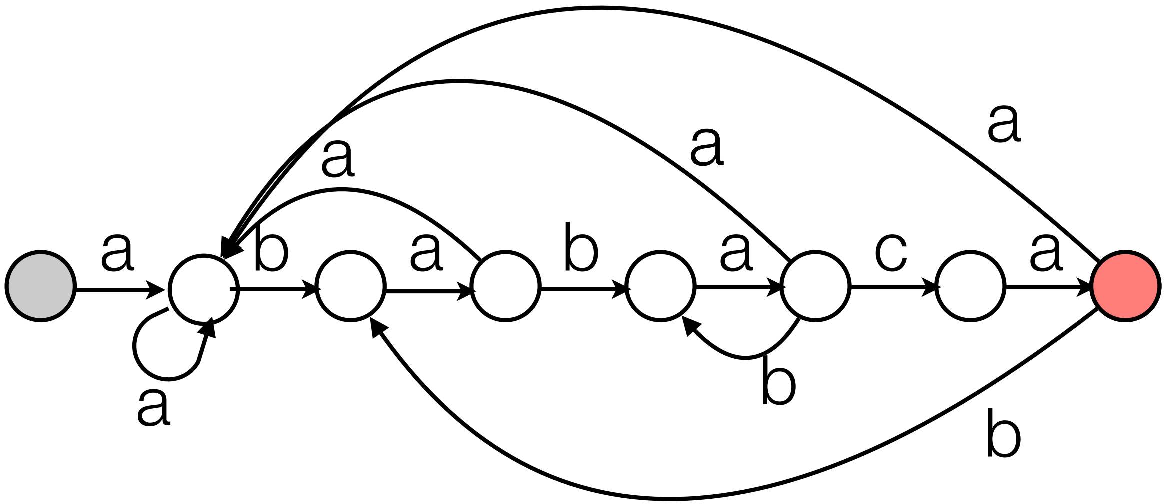



read ‘a’?

longest prefix of P that is a proper suffix of ‘ababacaa’ = ‘a’

# Finite Automaton Construction

- Finite automaton: alphabet Σ = {a,b,c}. P= ababaca.

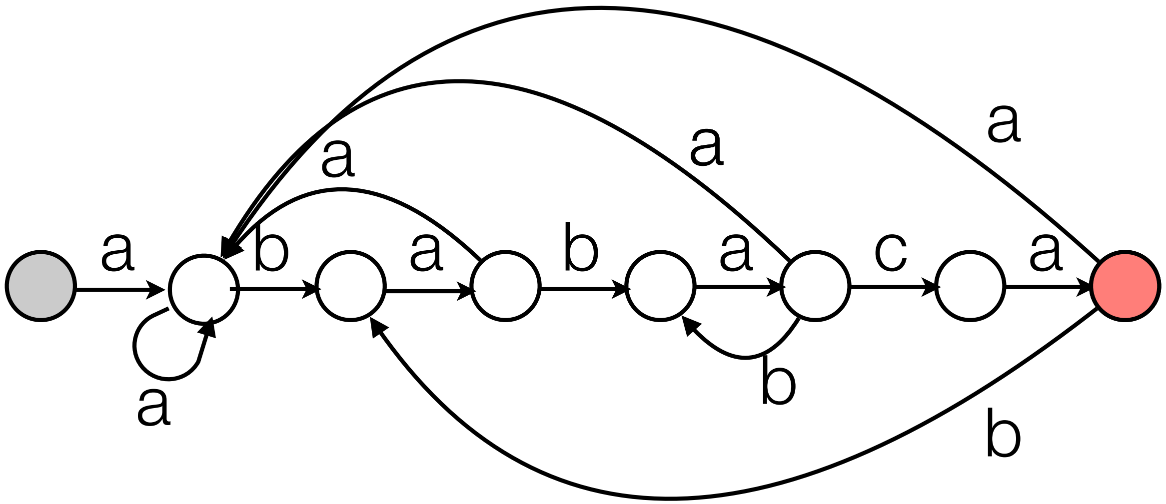



read ‘b’?

longest prefix of P that is a proper suffix of ‘ababacab’ = ‘ab’

# Finite Automaton Construction

- Finite automaton: alphabet $\Sigma$ = {a,b,c}. P= ababaca.

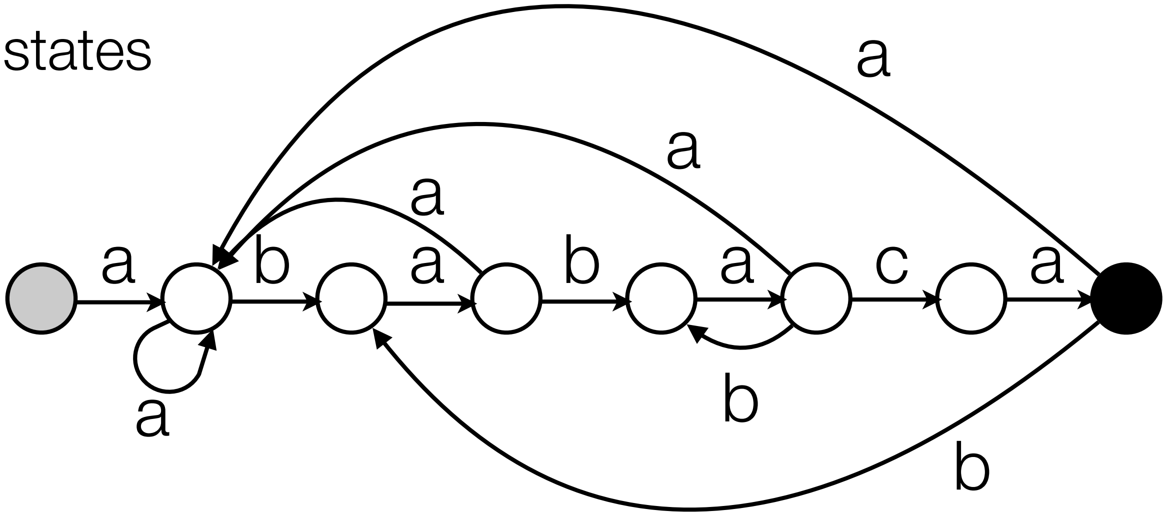



read ‘c’?

longest prefix of P that is a proper suffix of ‘ababacac’ = ‘ ’

# Finite Automaton

- Finite automaton:
  - Q: finite set of states
  - $q_0 \in Q$: start state
  - $A \subseteq Q$: set of accepting states
  - $\Sigma$: finite input alphabet
  - $\delta$: transition function

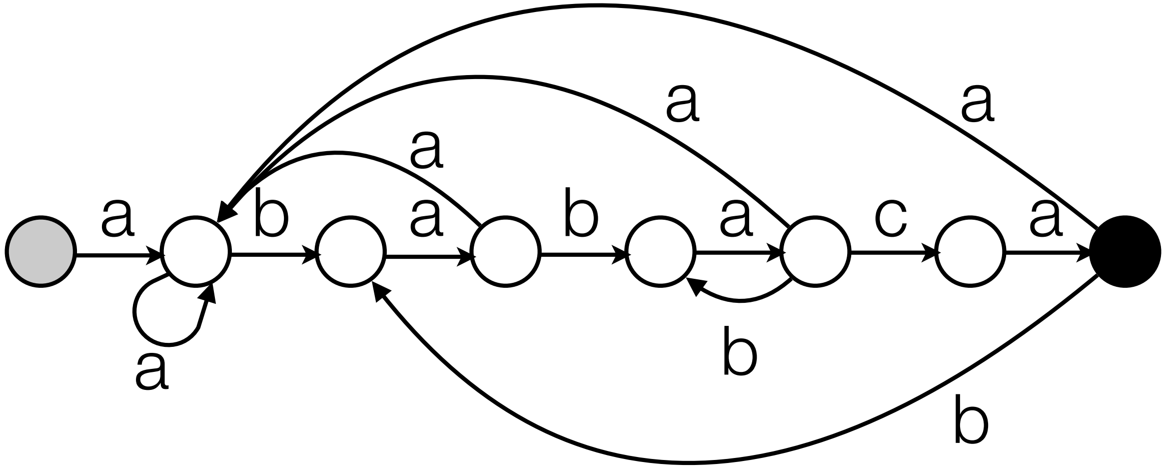



- Matching time: O(n)
- Preprocessing time: $O(m^3|\Sigma|)$. (Can be done in $O(m|\Sigma|)$).
- Total time: $O(n + m|\Sigma|)$

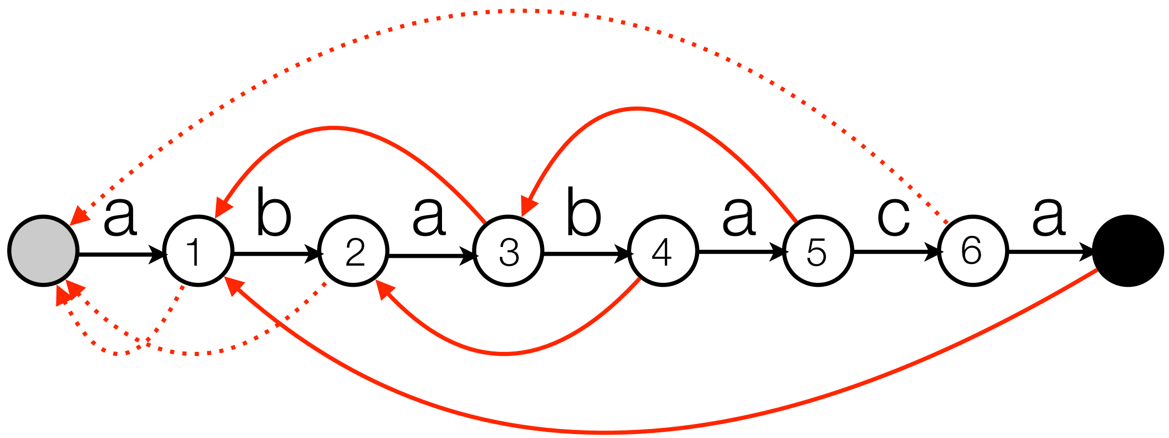
# KMP

# KMP

- Finite automaton: alphabet Σ = {a,b,c}. P = ababaca.

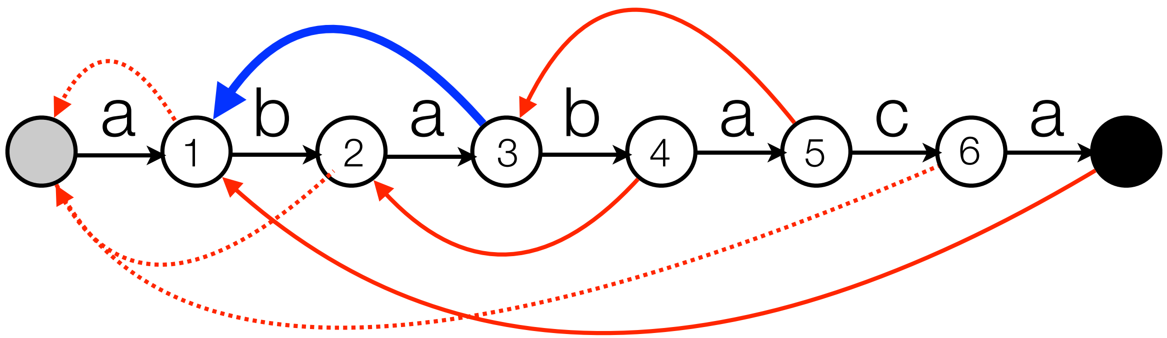



- KMP: Can be seen as finite automaton with *failure links*:



# KMP

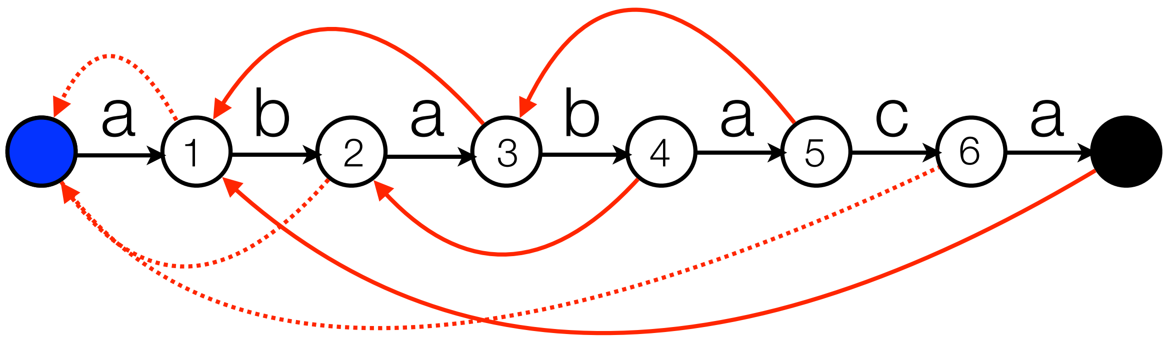
- KMP: Can be seen as finite automaton with *failure links*:
  - longest prefix of P that is a suffix of what we have *matched* until now (ignore the mismatched character).




longest prefix of P that is a proper suffix of ‘aba'

# KMP matching

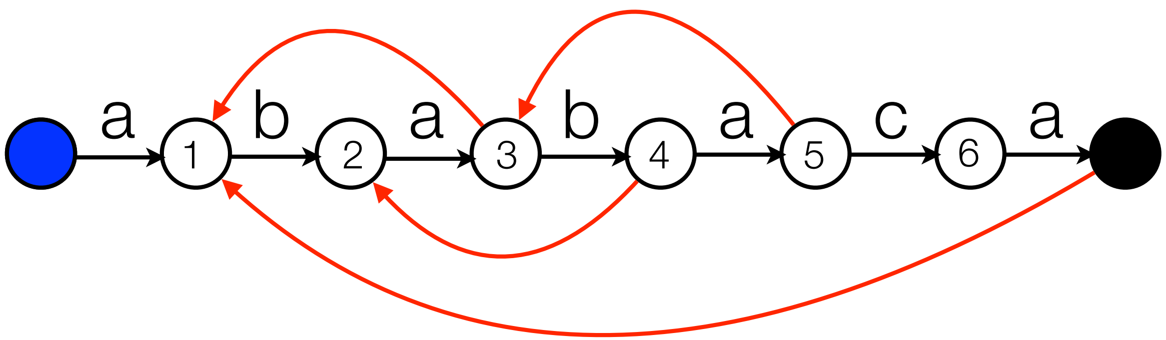
- KMP: Can be seen as finite automaton with *failure links*:
  - longest prefix of P that is a suffix of what we have *matched* until now.




T = b a c b a b a b a b a b a c a b

# KMP

- KMP: Can be seen as finite automaton with *failure links*:
  - longest prefix of P that is a proper suffix of what we have *matched* until now.
  - can follow several failure links when matching one character:




T = $\boxed{a}$ b a b a a

# KMP Analysis

- Analysis. $|T| = n$, $|P| = m$.
  - How many times can we follow a forward edge?
  - How many backward edges can we follow (compare to forward edges)?
  - Total number of edges we follow?
  - What else do we use time for?

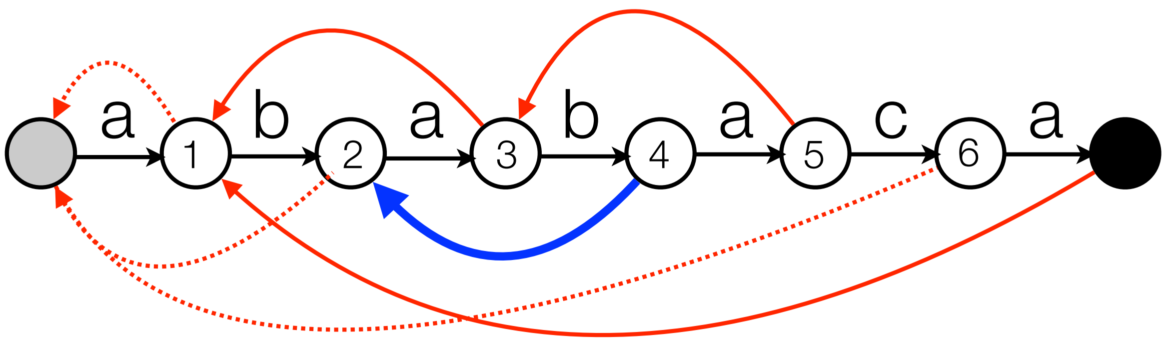
# KMP Analysis

- Lemma. The running time of KMP matching is $O(n)$.
  - Each time we follow a forward edge we read a new character of T.
  - #backward edges followed ≤ #forward edges followed ≤ n.
  - If in the start state and the character read in T does not match the forward edge, we stay there.
  - Total time = #non-matched characters in start state + #forward edges followed + #backward edges followed ≤ 2n.

# Computation of failure links

If we are in state j after reading T[1..i], then P[1..j] is the longest prefix of P that is a suffix of T[1..i].

- Failure link: longest prefix of P that is a proper suffix of what we have *matched* until now.




longest proper prefix of P that is a suffix of 'abab'

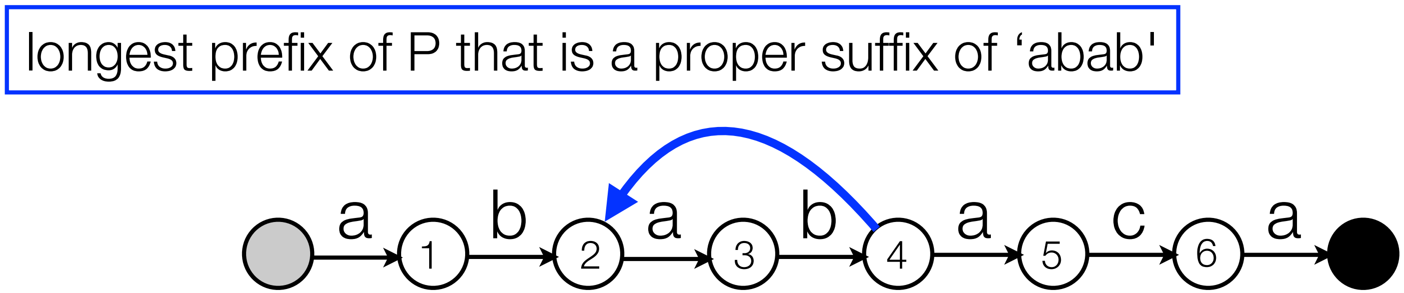
Matched until now: a b a b

a b a b a c a

# Computation of failure links

- Failure link: longest prefix of P that is a proper suffix of what we have *matched* until now.

- Computing failure links: Use KMP matching algorithm.

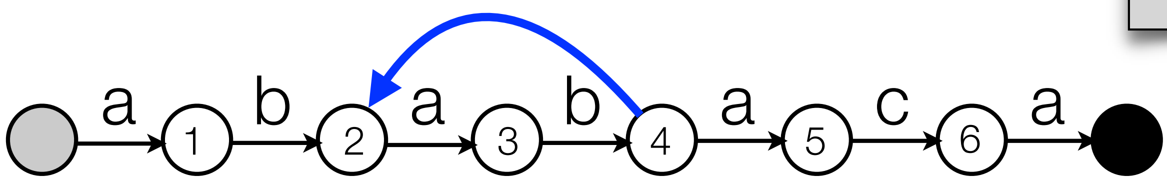


# Computation of failure links

- Failure link: longest prefix of P that is a proper suffix of what we have *matched* until now.
- Computing failure links: Use KMP matching algorithm.

If we are in state j after reading T[1..i], then P[1..j] is the longest prefix of P that is a suffix of T[1..i].

longest prefix of P that is a suffix of 'bab'



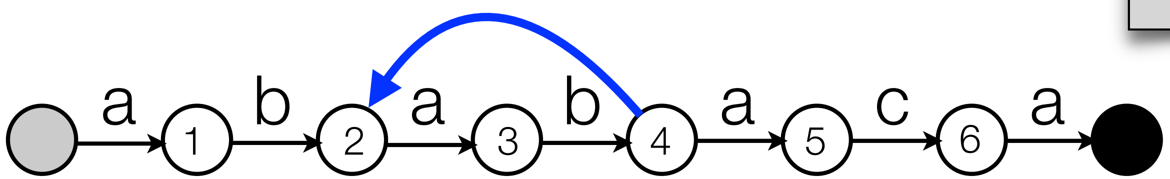
# Computation of failure links

- **Failure link:** longest prefix of P that is a proper suffix of what we have *matched* until now.

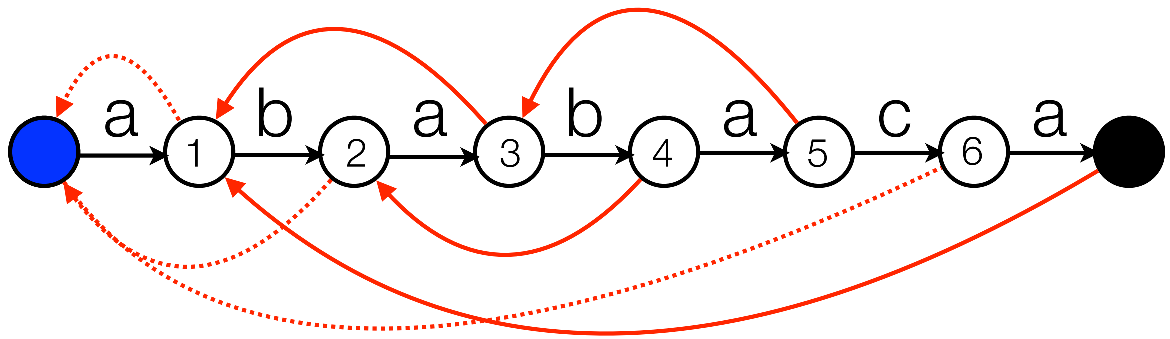
- **Computing failure links:** Use KMP matching algorithm.

If we are in state j after reading T[1..i], then P[1..j] is the longest prefix of P that is a suffix of T[1..i].

longest prefix of P that is a suffix of ‘bab'



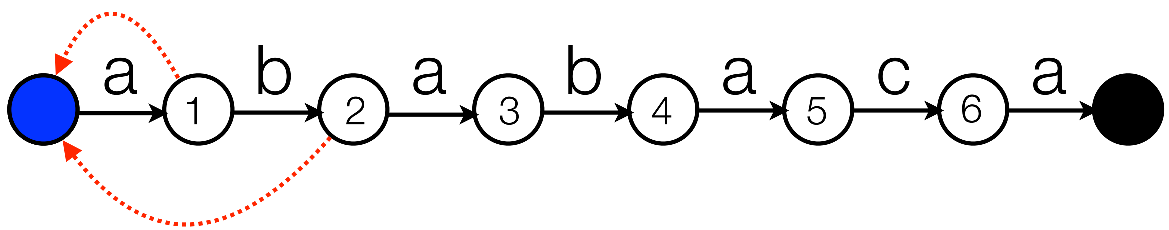

Can be found by using KMP to match ‘bab'



# Computation of failure links

- Computing failure links: As KMP matching algorithm (only need failure links that are already computed).

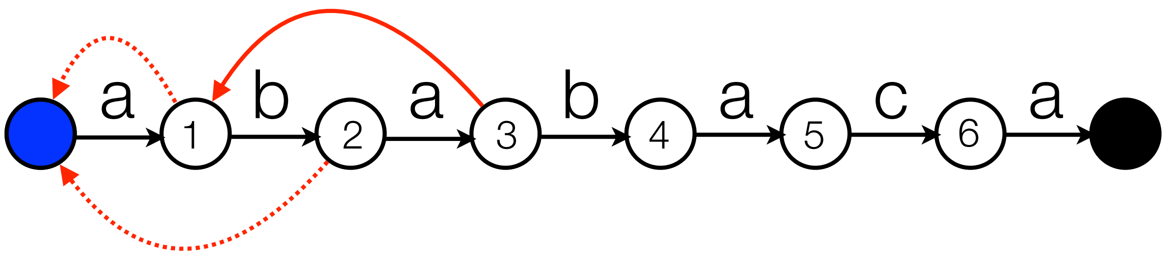
- Failure link: longest prefix of P that is a proper suffix of what we have *matched* until now.




T = $\boxed{b}$

# Computation of failure links

- Computing failure links: As KMP matching algorithm (only need failure links that are already computed).
- Failure link: longest prefix of P that is a proper suffix of what we have *matched* until now.

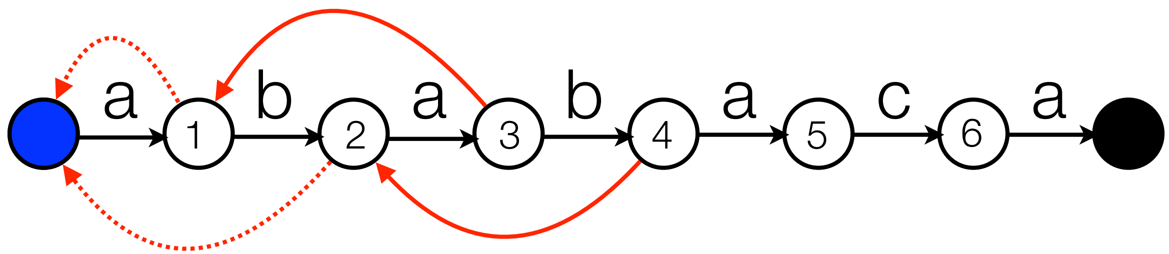



T = $\boxed{\text{b}}$ a

# Computation of failure links

- Computing failure links: As KMP matching algorithm (only need failure links that are already computed).

- Failure link: longest prefix of P that is a proper suffix of what we have *matched* until now.

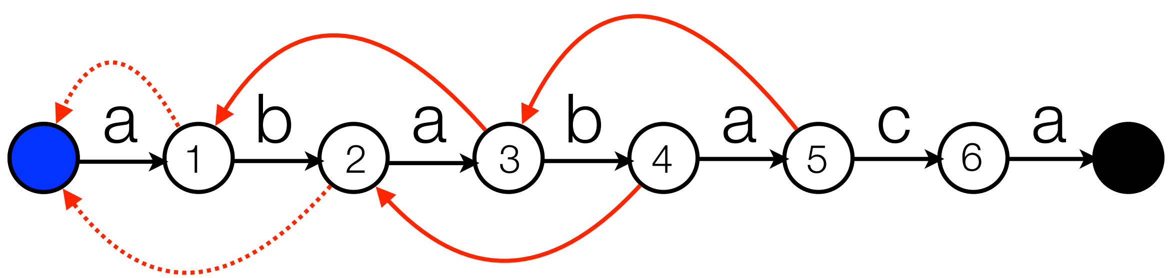



T = $\boxed{\text{b}}$ a b

# Computation of failure links

- Computing failure links: As KMP matching algorithm (only need failure links that are already computed).

- Failure link: longest prefix of P that is a proper suffix of what we have *matched* until now.

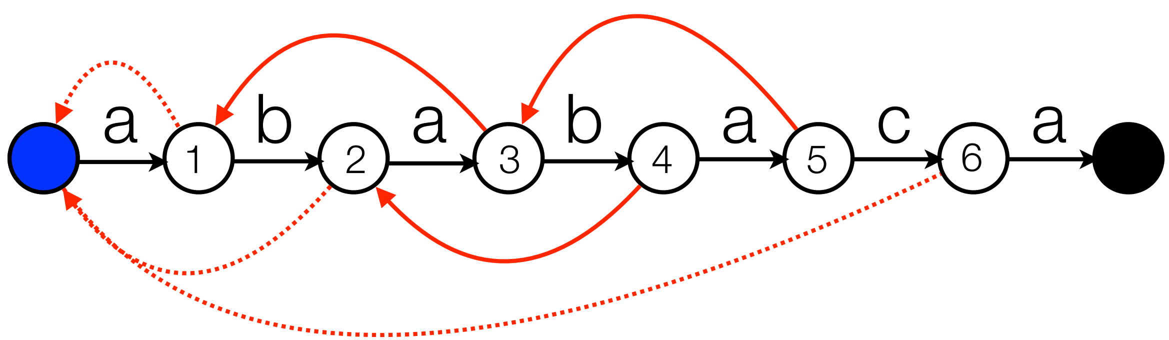



T = $\boxed{\text{b}}$ a b a

# Computation of failure links

- Computing failure links: As KMP matching algorithm (only need failure links that are already computed).

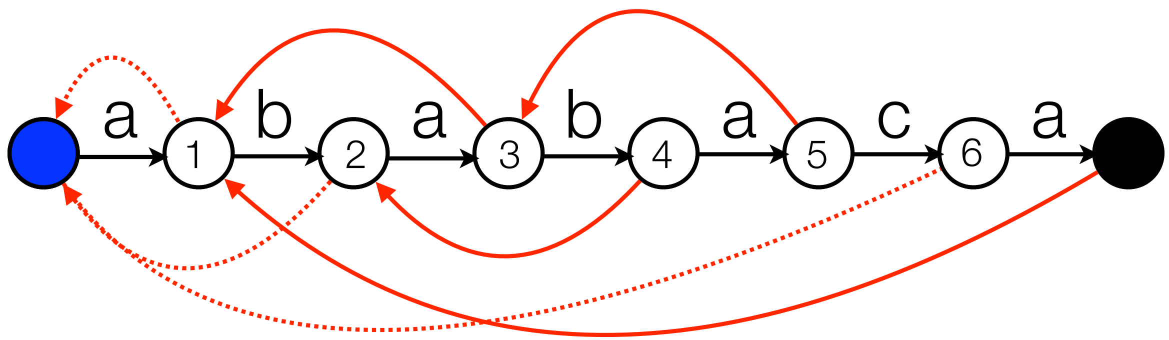
- Failure link: longest prefix of P that is a proper suffix of what we have *matched* until now.




T = $\boxed{b}$ a b a c

# Computation of failure links

- Computing failure links: As KMP matching algorithm (only need failure links that are already computed).
- Failure link: longest prefix of P that is a proper suffix of what we have *matched* until now.

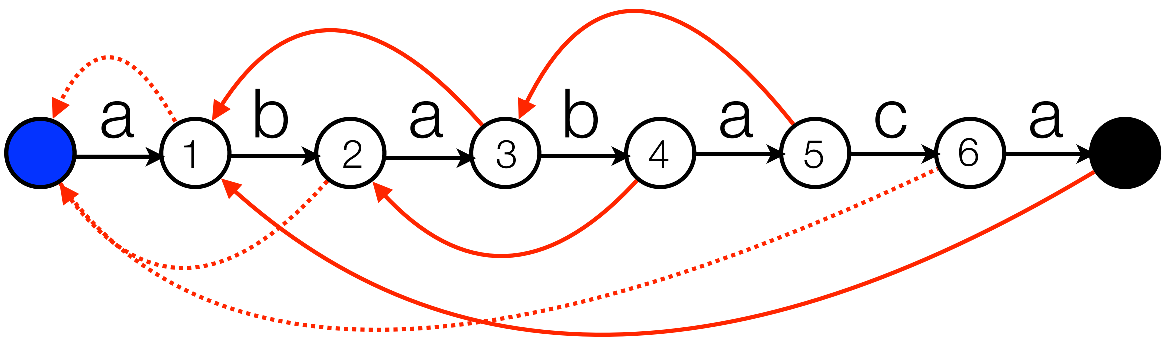



T = $\boxed{b}$ a b a c a

# Computation of failure links

- Computing failure links: As KMP matching algorithm (only need failure links that are already computed).

- Failure link: longest prefix of P that is a proper suffix of what we have *matched* until now.




1 2 3 4 5 6 7

P = [redacted] $\boxed{b}$ a b a c a

# KMP

- Computing π: As KMP matching algorithm (only need π values that are already computed).
- Running time: O(n + m):
  - Lemma. Total number of comparisons of characters in KMP is at most 2n.
  - Corollary. Total number of comparisons of characters in the preprocessing of KMP is at most 2m.

# KMP: the π array

- π array: A representation of the failure links.
- Takes up less space than pointers.

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| π[i] | 0 | 0 | 1 | 2 | 3 | 0 | 1 |