UNIVERSITÄT DES SAARLANDES

ZBI ZENTRUM FÜR BIOINFORMATIK

Exact Pattern Matching with Automata

Algorithms for Sequence Analysis

Sven Rahmann

Summer 2021

Recall the Pattern Matching Problem

```
alicewasbeginningtogetverytiredofsittingbyhersisteron
thebankandofhavingnothingtodoonceortwiceshehadpeepedi
ntothebookhersisterwasreadingbutithadnopicturesorconv
ersationsinitandwhatistheuseofabookthoughtalicewithou
tpicturesorconversation
```

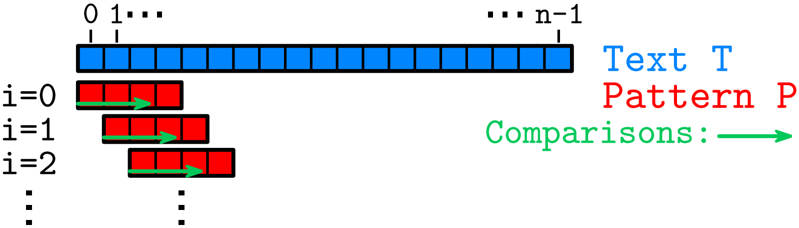
Task

Find all occurrences of a given string in another (longer) string.

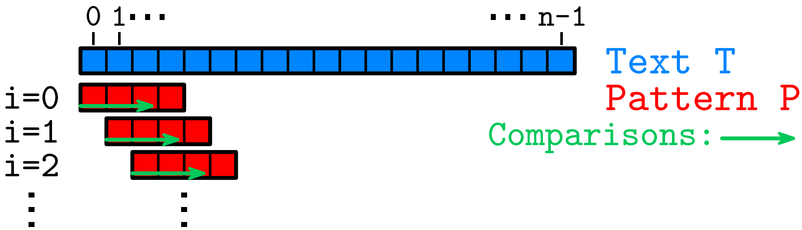
Goals

- As **fast** as possible (running time)
- As **easily** as possible (algorithm/implementation)

Recall the naïve algorithm



Recall the naïve algorithm



Ideas

- Can we shift window by more than one character?
  → Horspool algorithm (and others)
- We "touch" the same characters in T multiple times.
  Can we "reuse" information from preceeding comparisons?
  → Automata based algorithms (now)

Finite Automata Revisited

Deterministic Finite Automata (DFA)

Definition (DFA)

A **DFA** is a tuple $(Q, q_0, F, \Sigma, \delta)$, where

- Q is a finite set of **states**,
- $q_0 \in Q$ is a **start state**,
- $F \subset Q$ is a set of **accepting states**,
- Σ is an input **alphabet**, and
- $\delta : Q \times \Sigma \to Q$ is a **transition function**.

DFA – Example

- Q is a finite set of **states**,
- $q_0 \in Q$ is a **start state**,
- $F \subset Q$ is a set of **accepting states**,
- Σ is an input **alphabet**, and
- $\delta : Q \times \Sigma \to Q$ is a **transition function**.

Accept the strings over $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, where 4 divides the sum of the number of $\mathtt{a}$s and $\mathtt{b}$s:

Non-Deterministic Finite Automata (NFA)

Definition (NFA)

An **NFA** is a tuple $(Q, Q_0, F, \Sigma, \Delta)$, where

- Q is a finite set of **states**,
- $Q_0 \subset Q$ is a set of **start states**,
- $F \subset Q$ is a set of **accepting states**,
- Σ is an input **alphabet**, and
- $\Delta\colon Q \times \Sigma \to 2^Q$ is a (non-deterministic) **transition function**.

NFA – Example

- Q is a finite set of **states**,
- $Q_0 \subset Q$ is a set of **start states**,
- $F \subset Q$ is a set of **accepting states**,
- Σ is an input **alphabet**, and
- $\Delta\colon Q \times \Sigma \to 2^Q$ is a (non-deterministic) **transition function**.

Accept the strings over $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, where 3 or 4 divides the sum of the number of $\mathtt{a}$s and $\mathtt{b}$s:

Extending the Transition Function

Original NFA transition function: $\Delta\colon Q \times \Sigma \to 2^Q$

For notational convenience, we make the following definitions.

Extension to sets of states

- $\Delta(A, c) := \bigcup_{q \in A} \Delta(q, c)$ for a **set** of states A and $c \in \Sigma$.

Extension to strings

- $\Delta(A, \epsilon) := A$, where ϵ is the empty string, and
- $\Delta(A, xc) := \Delta(\Delta(A, x), c)$, where $x \in \Sigma^*$ and $c \in \Sigma$.

NFAs for Pattern Matching

NFA to Solve the Pattern Matching Problem

Goal

For given pattern $P \in \Sigma^*$, construct NFA that recognizes all strings $\Sigma^* P$.

NFA to Solve the Pattern Matching Problem

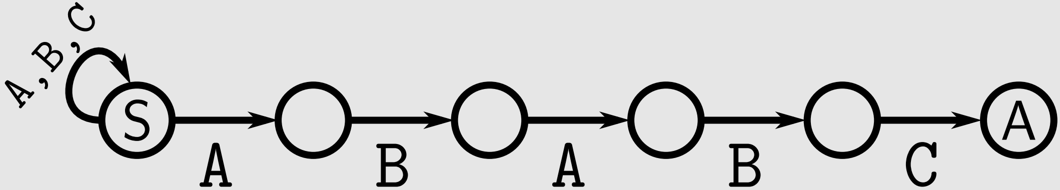
Goal

For given pattern $P \in \Sigma^*$, construct NFA that recognizes all strings $\Sigma^* P$.

Approach

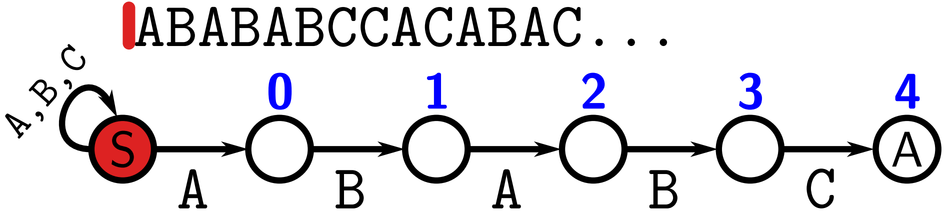
- "Linear chain" of states
- Start state remains always active

Example: $\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$ and $P = \mathtt{ABABC}$



Using our NFA

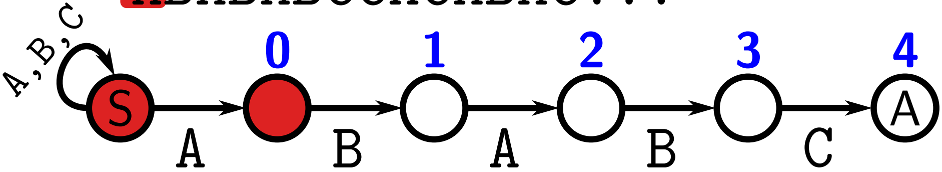
$\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$ and $P = \mathtt{ABABC}$



Using our NFA

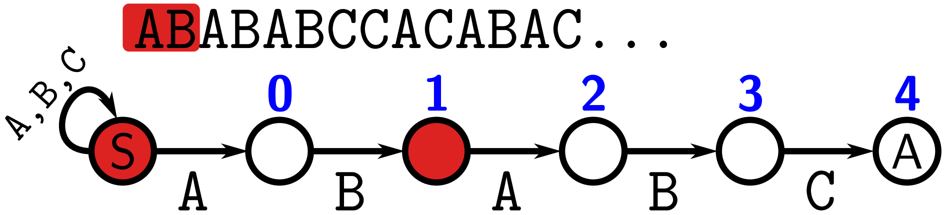
$\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$ and $P = \mathtt{ABABC}$

ABABABCCACABAC...



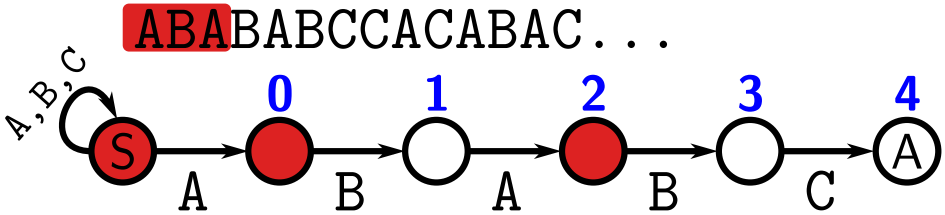
Using our NFA

$\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$ and $P = \mathtt{ABABC}$



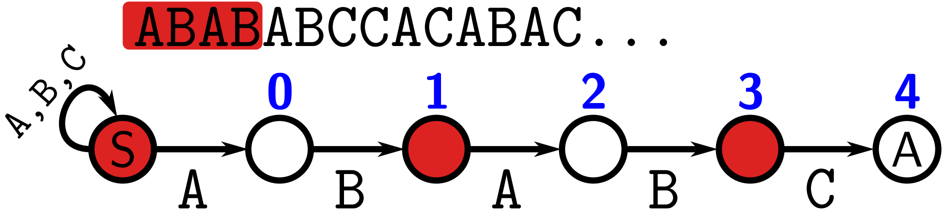
Using our NFA

$\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$ and $P = \mathtt{ABABC}$



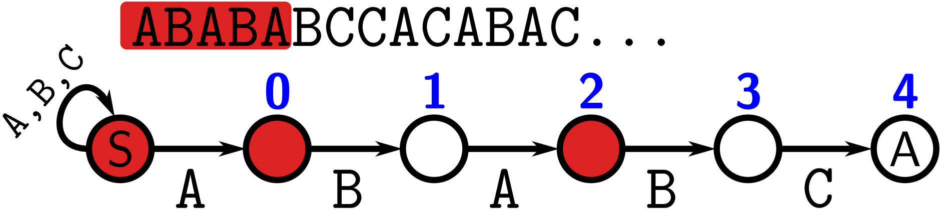
Using our NFA

$\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$ and $P = \mathtt{ABABC}$



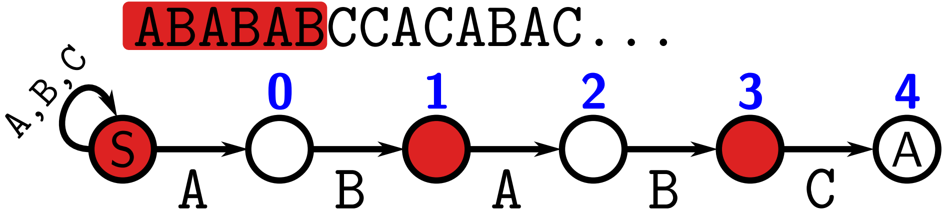
Using our NFA

$\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$ and $P = \mathtt{ABABC}$



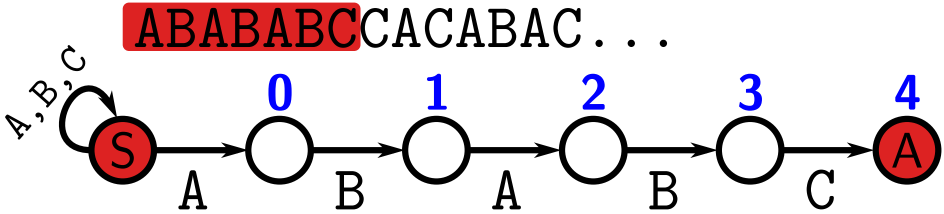
Using our NFA

$\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$ and $P = \mathtt{ABABC}$



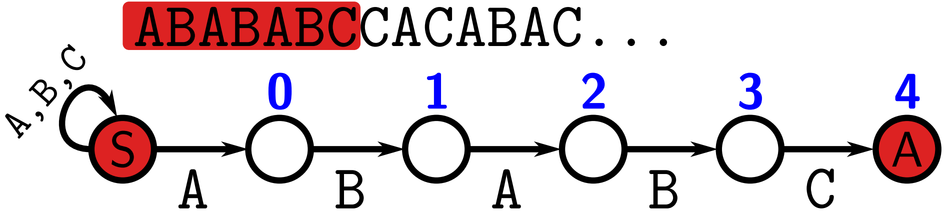
Using our NFA

$\Sigma = \{\texttt{A}, \texttt{B}, \texttt{C}\}$ and $P = \texttt{ABABC}$



Using our NFA

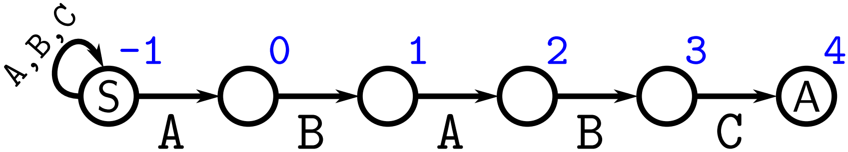
$\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$ and $P = \mathtt{ABABC}$



Things left to do

- Formally define this automaton,
- Give efficient implementation.

Pattern Matching NFA (Formal)



- state set $Q = \{-1, 0, \dots, m-1\}$, where $m = |P|$
- start states $Q_0 = \{-1\}$
- accepting states $F = \{m-1\}$
- transition function Δ:

For $q = -1$: $\Delta(-1, c) = \begin{cases} \{-1, 0\} & \text{if } c = P[0], \\ \{-1\} & \text{otherwise.} \end{cases}$

For $q \in \{0, \dots, m-2\}$: $\Delta(q, c) = \begin{cases} \{q+1\} & \text{if } c = P[q+1], \\ \emptyset & \text{otherwise.} \end{cases}$

For $q = m-1$: $\Delta(m-1, c) = \emptyset$

Correctness

Lemma: NFA state set invariant

Let $A \subset Q$ be a set of active states of our NFA. Then, $q \in A \setminus \{-1\}$ iff the last $q+1$ read characters equal the patterns prefix $P[\dots q]$. In particular, state $|P|-1$ is active iff the last $|P|$ characters equal the full pattern.

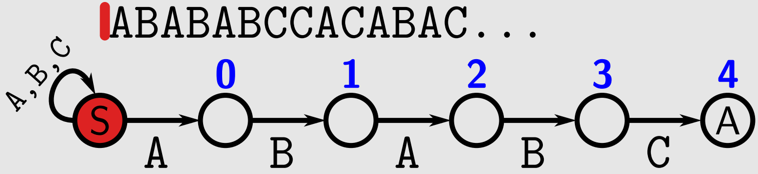
Proof: Follows directly from the NFA definition.

Theorem: Correctness of Pattern Matching NFA

The pattern matching NFA for pattern P accepts exactly the language $\Sigma^* P$.

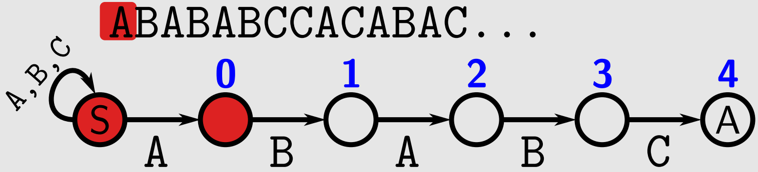
Proof: Follows immediately from the above lemma.

Efficient Implementation: The Shift-And Algorithm



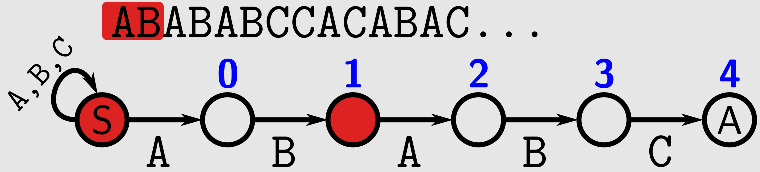
D:	0	0	0	0	0
mask(A):	1	0	1	0	0
mask(B):	0	1	0	1	0
mask(C):	0	0	0	0	1

Efficient Implementation: The Shift-And Algorithm



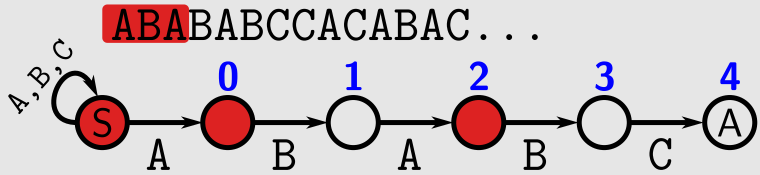
D:	1	0	0	0	0
mask(A):	1	0	1	0	0
mask(B):	0	1	0	1	0
mask(C):	0	0	0	0	1

Efficient Implementation: The Shift-And Algorithm



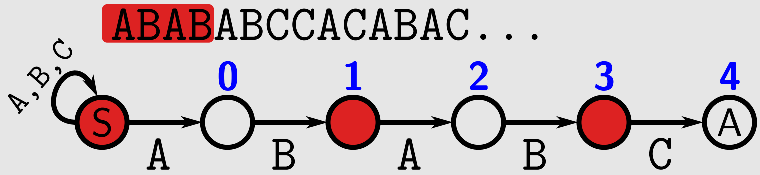
	0	1	2	3	4
D:	0	1	0	0	0
$mask(\mathtt{A})$:	1	0	1	0	0
$mask(\mathtt{B})$:	0	1	0	1	0
$mask(\mathtt{C})$:	0	0	0	0	1

Efficient Implementation: The Shift-And Algorithm



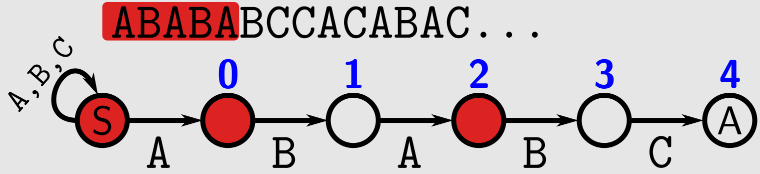
D:	1	0	1	0	0
mask(A):	1	0	1	0	0
mask(B):	0	1	0	1	0
mask(C):	0	0	0	0	1

Efficient Implementation: The Shift-And Algorithm



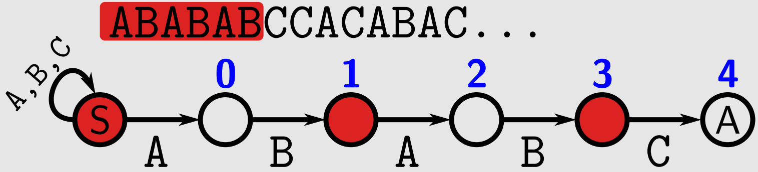
D:	0	1	0	1	0
$mask(\mathtt{A})$:	1	0	1	0	0
$mask(\mathtt{B})$:	0	1	0	1	0
$mask(\mathtt{C})$:	0	0	0	0	1

Efficient Implementation: The Shift-And Algorithm



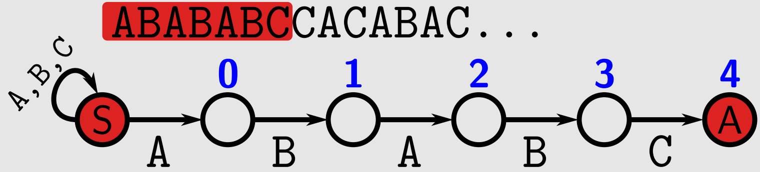
D:	1	0	1	0	0
mask(A):	1	0	1	0	0
mask(B):	0	1	0	1	0
mask(C):	0	0	0	0	1

Efficient Implementation: The Shift-And Algorithm



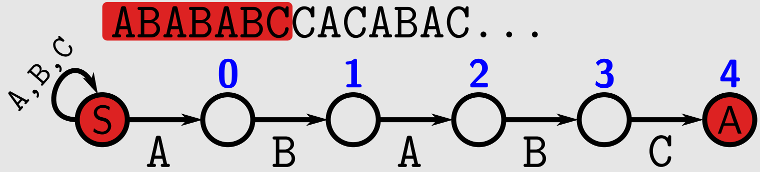
D:	0	1	0	1	0
mask(A):	1	0	1	0	0
mask(B):	0	1	0	1	0
mask(C):	0	0	0	0	1

Efficient Implementation: The Shift-And Algorithm



D:	0	0	0	0	1
mask(A):	1	0	1	0	0
mask(B):	0	1	0	1	0
mask(C):	0	0	0	0	1

Efficient Implementation: The Shift-And Algorithm

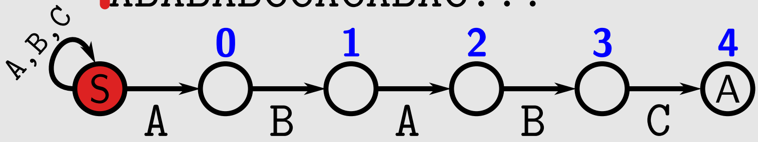


D:	0	0	0	0	1
mask(A):	1	0	1	0	0
mask(B):	0	1	0	1	0
mask(C):	0	0	0	0	1

$$D \leftarrow ((D \ll 1) \mid 1) \;\&\; \mathrm{mask}(c)$$

Efficient Implementation: The Shift-And Algorithm

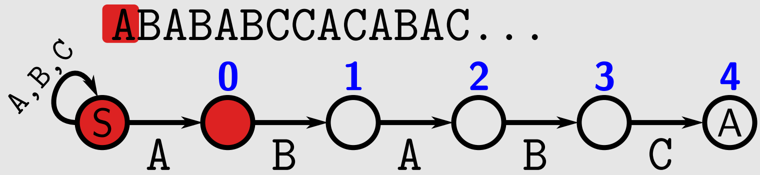
ABABABCCACABAC...



D:	0	0	0	0	0
$mask(\texttt{A})$:	1	0	1	0	0
$mask(\texttt{B})$:	0	1	0	1	0
$mask(\texttt{C})$:	0	0	0	0	1

$$D \leftarrow ((D \ll 1) \mid 1) \ \& \ \mathrm{mask}(c)$$

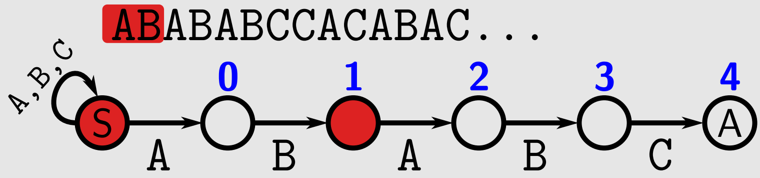
Efficient Implementation: The Shift-And Algorithm



D:	1	0	0	0	0
$mask(\mathtt{A})$:	1	0	1	0	0
$mask(\mathtt{B})$:	0	1	0	1	0
$mask(\mathtt{C})$:	0	0	0	0	1

$D \leftarrow \big((D \ll 1) \mid 1\big)\ \&\ \mathrm{mask}(c)$

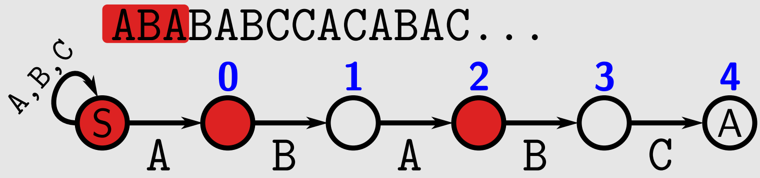
Efficient Implementation: The Shift-And Algorithm



D:	0	1	0	0	0
mask(A):	1	0	1	0	0
mask(B):	0	1	0	1	0
mask(C):	0	0	0	0	1

$$D \leftarrow ((D \ll 1) \mid 1)\ \&\ \text{mask}(c)$$

Efficient Implementation: The Shift-And Algorithm

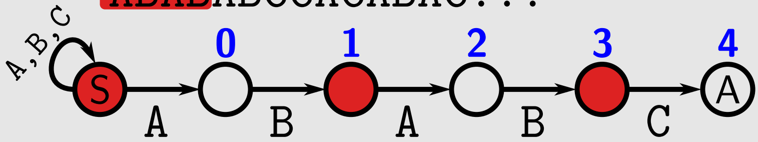


D:	1	0	1	0	0
mask(A):	1	0	1	0	0
mask(B):	0	1	0	1	0
mask(C):	0	0	0	0	1

$$D \leftarrow \big((D \ll 1) \mid 1\big) \;\&\; \mathrm{mask}(c)$$

Efficient Implementation: The Shift-And Algorithm

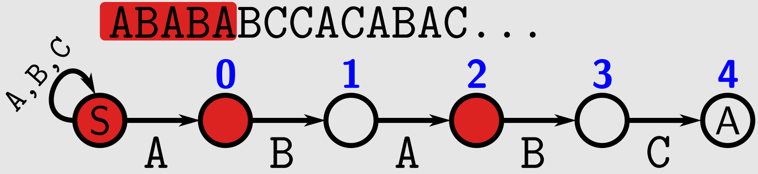
ABABABCCACABAC...



D:	0	1	0	1	0
$mask(\texttt{A})$:	1	0	1	0	0
$mask(\texttt{B})$:	0	1	0	1	0
$mask(\texttt{C})$:	0	0	0	0	1

$$D \leftarrow ((D \ll 1) \mid 1) \ \& \ \text{mask}(c)$$

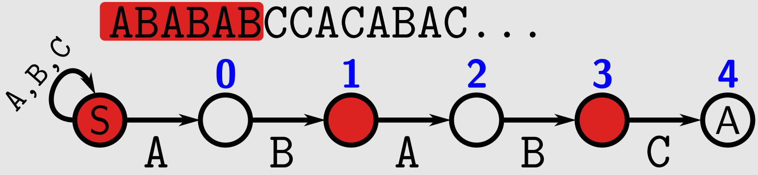
Efficient Implementation: The Shift-And Algorithm



D:	1	0	1	0	0
$mask(\mathtt{A})$:	1	0	1	0	0
$mask(\mathtt{B})$:	0	1	0	1	0
$mask(\mathtt{C})$:	0	0	0	0	1

$$D \leftarrow \big((D \ll 1) \mid 1\big)\ \&\ \mathrm{mask}(c)$$

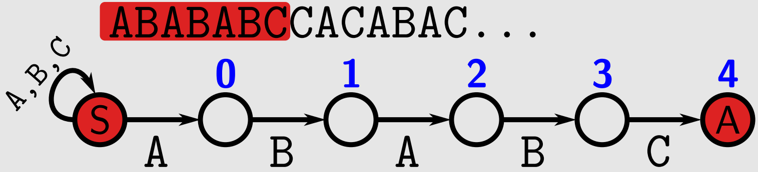
Efficient Implementation: The Shift-And Algorithm



D:	0	1	0	1	0
$mask(\mathtt{A})$:	1	0	1	0	0
$mask(\mathtt{B})$:	0	1	0	1	0
$mask(\mathtt{C})$:	0	0	0	0	1

$$D \leftarrow \big((D \ll 1) \mid 1\big) \;\&\; \mathrm{mask}(c)$$

Efficient Implementation: The Shift-And Algorithm



D:	0	0	0	0	1
$mask(\mathtt{A})$:	1	0	1	0	0
$mask(\mathtt{B})$:	0	1	0	1	0
$mask(\mathtt{C})$:	0	0	0	0	1

$$D \leftarrow ((D \ll 1) \mid 1) \;\&\; \mathrm{mask}(c)$$

Code for Shift-And Algorithm

```
def ShiftAnd(P, T):
    m = len(P)
    masks = dict() # empty dictionary
    bit = 1
    for c in P:
        if c not in masks: masks[c] = 0
        masks[c] = masks[c] | bit
        bit = bit * 2
    accept_state = bit // 2
    D = 0  # bit-mask of active states
    i = 0
    for c in T:
        D = ((D << 1) | 1) & masks[c]
        if (D & accept_state) != 0:
            yield i
        i += 1
```

Running time of Shift-And Algorithm

- m: Pattern length
- n: Text length
- w: Machine register width

Running time

If $m < w$, then the shift-and algorithm runs in $O(m+n)$ time.
Generally (i.e. when m/w is not constant), it takes $O(m + nm/w)$ time.

Conclusions

- Fast when pattern fits into one machine word
- Running time independent of how similar text and pattern are.
- Running time independent of alphabet size.

DFA-based Pattern Matching

Reminder: How to Turn an NFA into a DFA

Definition: equivalence of automata

Two automata are **equivalent** if they accept the same sets of words (languages).

NFA vs. DFA

- **NFA:** Many states can be active at the same time
- **DFA:** Only one state active at any given time

Subset construction (NFA $\rightarrow$ DFA)

- **Idea:** create one DFA state for every **subset** of NFA states
- We can omit states that are **unreachable**

Example: Subset Construction (NFA $\to$ DFA)

Accept the strings over $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, where 3 or 4 divides the sum of the number of $\mathtt{a}$s and $\mathtt{b}$s:

Subset Construction: How Many DFA States?

In general

Subset construction leads to **exponential blow-up** in the number of states:
$|Q|$ NFA states turn into $2^{|Q|}$ DFA states.

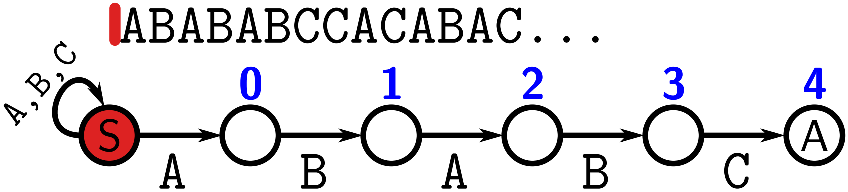
In practice

Blow-up often does not happen (many states are **unreachable**).

For our pattern matching automata

DFA always has the **same** number of states as **NFA**.

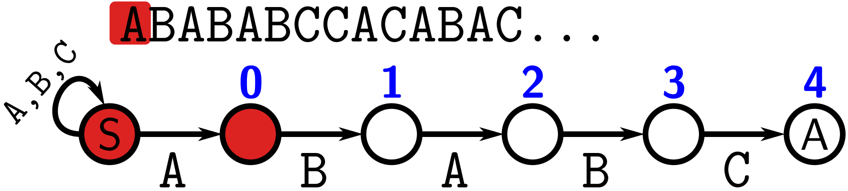
Reachable State Sets in NFA



Lemma (from previous lecture): NFA state set invariant

Let $A \subset Q$ be a set of active states of our NFA. Then, $q \in A \setminus \{-1\}$ iff the last $q+1$ read characters equal the pattern's prefix $P[\ldots q]$. In particular, state $|P|-1$ is active iff the last $|P|$ characters equal the full pattern.

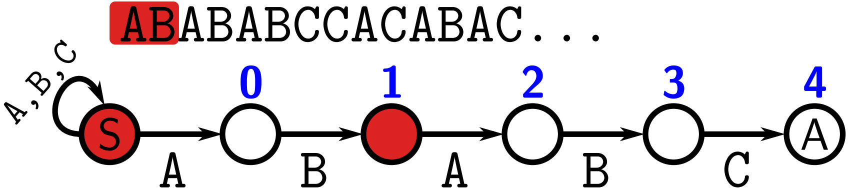
Reachable State Sets in NFA



Lemma (from previous lecture): NFA state set invariant

Let $A \subset Q$ be a set of active states of our NFA. Then, $q \in A \setminus \{-1\}$ iff the last $q+1$ read characters equal the pattern's prefix $P[\ldots q]$. In particular, state $|P|-1$ is active iff the last $|P|$ characters equal the full pattern.

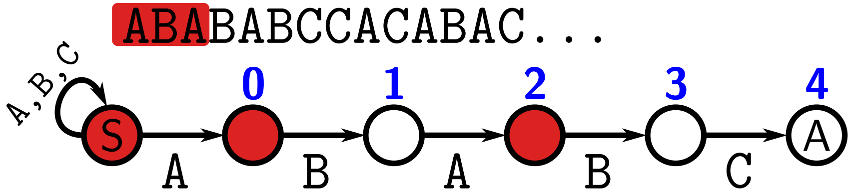
Reachable State Sets in NFA



Lemma (from previous lecture): NFA state set invariant

Let $A \subset Q$ be a set of active states of our NFA. Then, $q \in A \setminus \{-1\}$ iff the last $q+1$ read characters equal the pattern's prefix $P[\ldots q]$. In particular, state $|P|-1$ is active iff the last $|P|$ characters equal the full pattern.

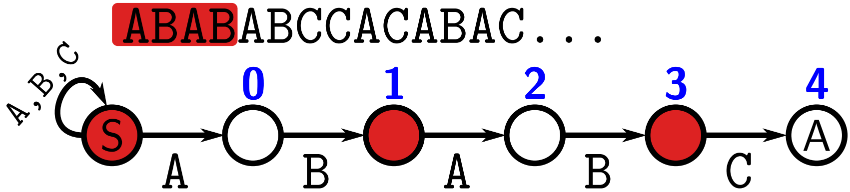
Reachable State Sets in NFA



Lemma (from previous lecture): NFA state set invariant

Let $A \subset Q$ be a set of active states of our NFA. Then, $q \in A \setminus \{-1\}$ iff the last $q + 1$ read characters equal the pattern's prefix $P[\ldots q]$. In particular, state $|P| - 1$ is active iff the last $|P|$ characters equal the full pattern.

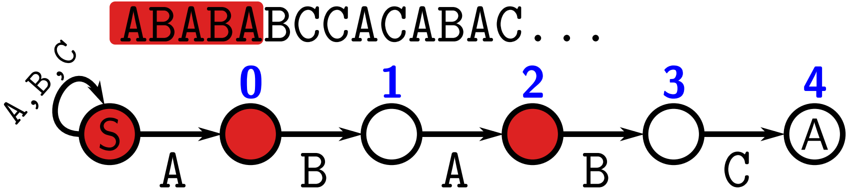
Reachable State Sets in NFA



Lemma (from previous lecture): NFA state set invariant

Let $A \subset Q$ be a set of active states of our NFA. Then, $q \in A \setminus \{-1\}$ iff the last $q+1$ read characters equal the pattern's prefix $P[\ldots q]$. In particular, state $|P|-1$ is active iff the last $|P|$ characters equal the full pattern.

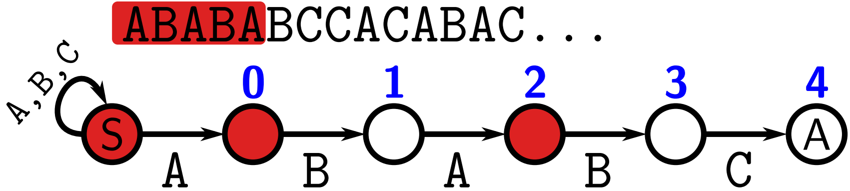
Reachable State Sets in NFA



Lemma (from previous lecture): NFA state set invariant

Let $A \subset Q$ be a set of active states of our NFA. Then, $q \in A \setminus \{-1\}$ iff the last $q+1$ read characters equal the pattern's prefix $P[\ldots q]$. In particular, state $|P|-1$ is active iff the last $|P|$ characters equal the full pattern.

Reachable State Sets in NFA



Lemma (from previous lecture): NFA state set invariant

Let $A \subset Q$ be a set of active states of our NFA. Then, $q \in A \setminus \{-1\}$ iff the last $q+1$ read characters equal the pattern's prefix $P[\ldots q]$. In particular, state $|P|-1$ is active iff the last $|P|$ characters equal the full pattern.

Lemma

Let A be the set of active states of a pattern matching NFA. Now, let $a^* := \max A$. Then, A is completely determined by a^*.

Consequences of State Set Lemma

Lemma (from previous slide)

Let A be the set of active states of a pattern matching NFA. Now, let $a^* := \max A$. Then, A is completely determined by a^*.

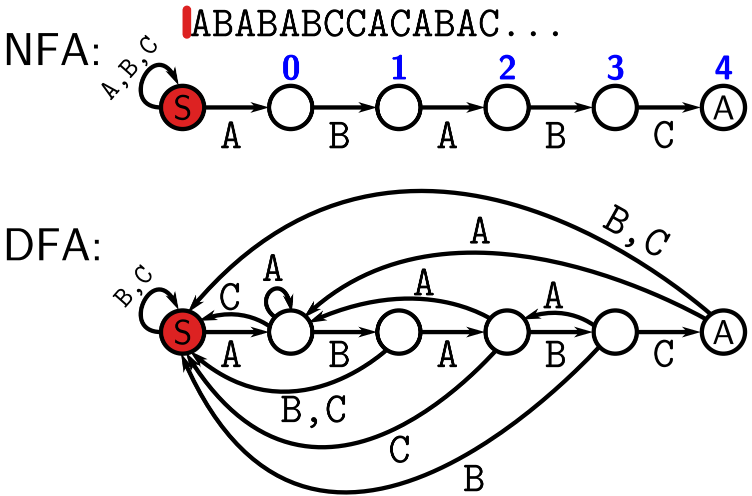
Consequences

For each possible a^* from -1 to $m-1$, there is exactly one reachable set of active states.
$\Rightarrow$ there are exactly $m+1$ possible sets of active states.
$\Rightarrow$ there is an equivalent DFA with $m+1$ states.

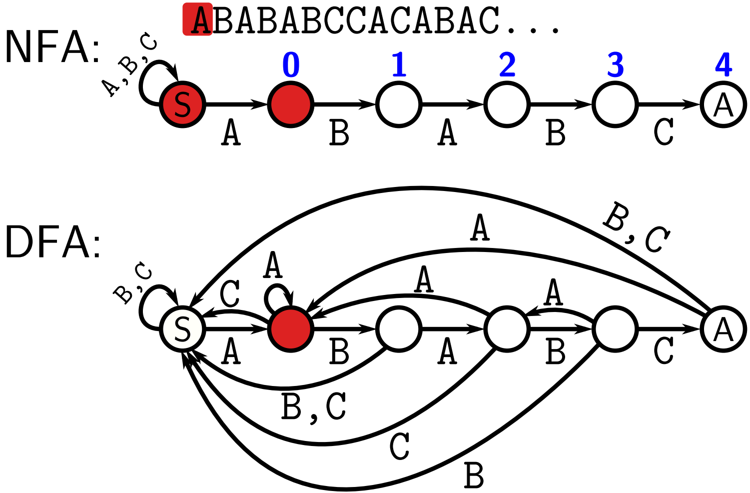
DFA state set

- We use same state set $Q = \{-1, \ldots, m-1\}$ for DFAs.
- Being in state $q \in Q$ in DFA $\Longleftrightarrow$ NFA has active states set A with $\max A = q$.

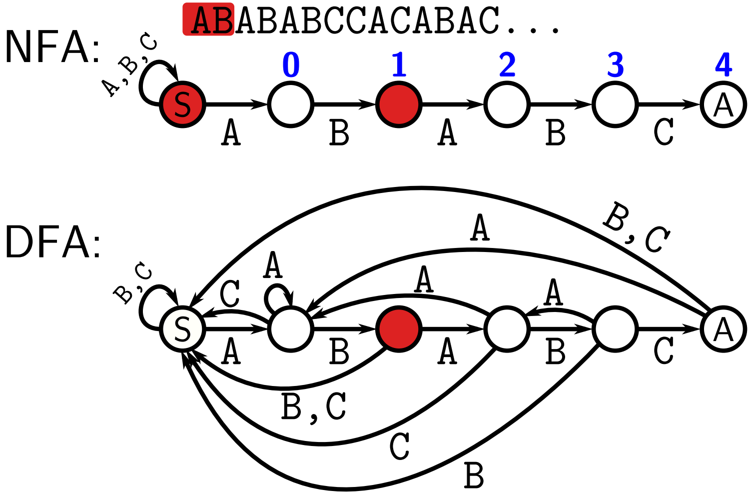
DFA-based vs. NFA-based Pattern Matching



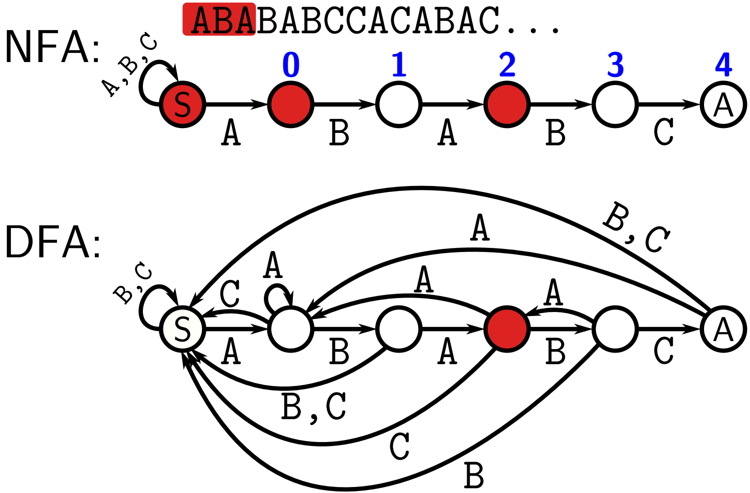
DFA-based vs. NFA-based Pattern Matching



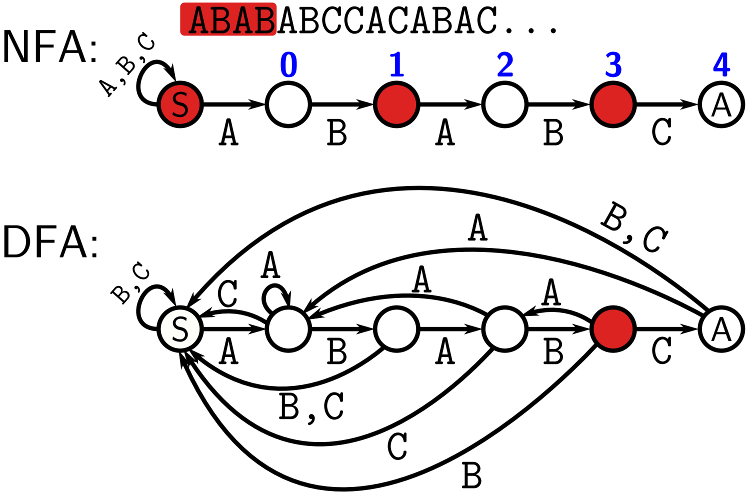
DFA-based vs. NFA-based Pattern Matching



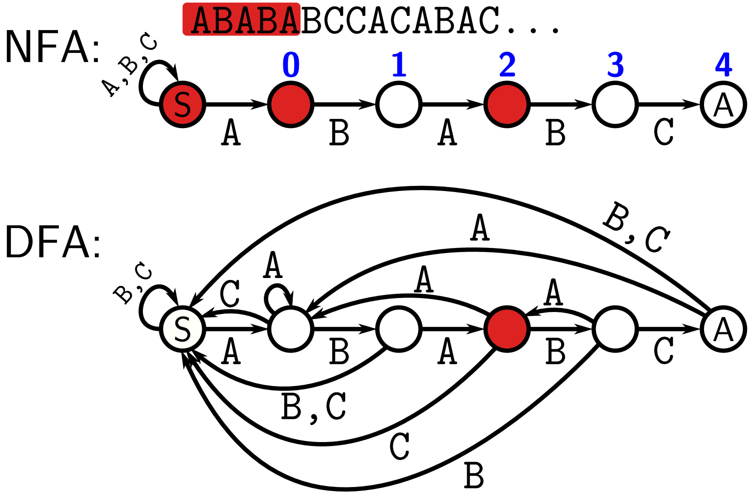
DFA-based vs. NFA-based Pattern Matching



DFA-based vs. NFA-based Pattern Matching



DFA-based vs. NFA-based Pattern Matching



DFA-based Pattern Matching: Overview

Algorithm overview

1. Construct pattern matching NFA in $O(m)$ time.
2. Construct DFA by computing the set of active states for every possible a^* in $O(m^2)$ time.
3. Use DFA for searching text in $O(n)$ time.

DFA-based Pattern Matching: Overview

Algorithm overview

1. Construct pattern matching NFA in $O(m)$ time.
2. Construct DFA by computing the set of active states for every possible a^* in $O(m^2)$ time.
3. Use DFA for searching text in $O(n)$ time.

Resulting **total time**: $O(m^2 + n)$

DFA-based Pattern Matching: Code

```
def DFA_with_delta(m, delta, T):
    q = -1
    for i in range(len(T)):
        q = delta(q, T[i])
        if q == m - 1:
            yield (i-m+1, i+1)

def DFA(P, T):
    delta = DFA_delta_table(P)
    return DFA_with_delta(len(P), delta, T)
```

DFA-based Pattern Matching: Code

```
def DFA_with_delta(m, delta, T):
    q = -1
    for i in range(len(T)):
        q = delta(q, T[i])
        if q == m - 1:
            yield (i-m+1, i+1)

def DFA(P, T):
    delta = DFA_delta_table(P)
    return DFA_with_delta(len(P), delta, T)
```

Summary

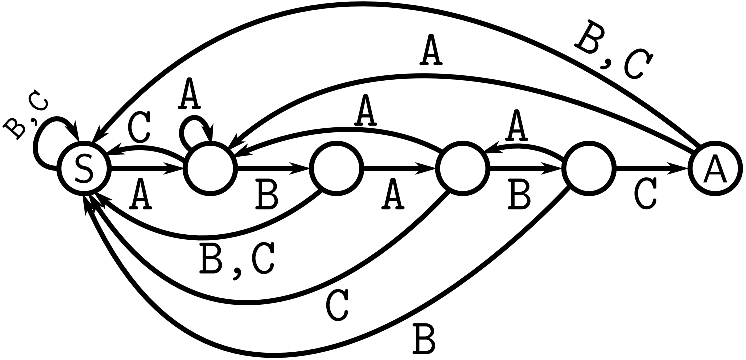
- Total time: $O(m^2 + n)$
- How do we get to $O(m + n)$?

Knuth-Morris-Pratt Algorithm

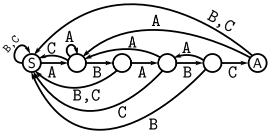
How to Compute the DFA Transition Function?

DFA transition function δ

- Matching character: just move right (easy)
- Non-matching character: move left (but how far?)



Example: Construction of DFA Transition Table (ABABC)



Using the *lps*-Table to Generate δ

Given: state q and character c

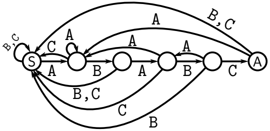
Approach to compute $\delta(q, c)$

- If $c = P[q+1]$, then $q \mapsto q+1$
- If not, try again for q', where q' is "the next NFA state that would be active" (if this was an NFA and not a DFA)

The lps-function: $lps : \{0, \ldots, m-1\} \to \mathbb{N}$

- Recall: state q corresponds to prefix $P[\ldots q]$ of length $q+1$
- $lps(q)$ is the length of the **l**ongest **p**refix of P that is a true **s**uffix of $P[\ldots q]$.
- Then, we can get q' through $q' = lps(q) - 1$.

Example: *lps*-Table



Using lps to Compute δ

```
1 def DFA_delta_lps(q, c, P, lps):
2     """state q, character c, pattern P, lps function/table"""
3     m = len(P)
4     while q == m-1 or (P[q+1] != c and q > -1):
5         q = lps[q] - 1
6     if P[q+1] == c:  q += 1
7     return q
```

Running time analysis

- Still takes $O(m)$ time in the worst case (`while` loop in line 4), leading to $O(mn)$ time for pattern matching over the whole text.

Using lps to Compute δ

```
1 def DFA_delta_lps(q, c, P, lps):
2     """state q, character c, pattern P, lps function/table"""
3     m = len(P)
4     while q == m-1 or (P[q+1] != c and q > -1):
5         q = lps[q] - 1
6     if P[q+1] == c:  q += 1
7     return q
```

Running time analysis

- Still takes $O(m)$ time in the worst case (`while` loop in line 4), leading to $O(mn)$ time for pattern matching over the whole text.
- But m iterations cannot happen for all n text characters! **Amortized analysis**: How many times can lines 4, 5 be executed in total?

Using lps to Compute δ

```
1 def DFA_delta_lps(q, c, P, lps):
2     """state q, character c, pattern P, lps function/table"""
3     m = len(P)
4     while q == m-1 or (P[q+1] != c and q > -1):
5         q = lps[q] - 1
6     if P[q+1] == c:  q += 1
7     return q
```

Running time analysis

- Still takes $O(m)$ time in the worst case (`while` loop in line 4), leading to $O(mn)$ time for pattern matching over the whole text.
- But m iterations cannot happen for all n text characters! **Amortized analysis**: How many times can lines 4, 5 be executed in total?
- Line 5 decreases q, but q cannot drop below -1. Only line 6 can ever increase q, at most once per iteration.

Knuth-Morris-Pratt Algorithm

```
def DFA_delta_lps(q, c, P, lps):
    """state q, character c, pattern P, lps function/table"""
    m = len(P)
    while q == m-1 or (P[q+1] != c and q > -1):
        q = lps[q] - 1
    if P[q+1] == c:  q += 1
    return q
```

```
def KMP(P, T):
    lps = compute_lps(P)
    m, q = len(P), -1
    for i in range(len(T)):
        q = DFA_delta_lps(q, T[i], P, lps)
        if q == m - 1: yield i
```

Running time: $O(n+m)$ since `DFA_delta_lps` takes **amortized** constant time.

Computing the lps-Table

```python
def compute_lps(P):
    m = len(P)
    q = -1
    lps = [0] * m  # lps[0] = 0 is correct
    for i in range(1, m):
        while q > -1 and P[q+1] != P[i]:
            q = lps[q] - 1
        if P[q+1] == P[i]:  q += 1
        # Invariant (Q) holds here
        lps[i] = q+1
    return lps
```

Invariant (Q): $q = \max\left\{k < i : P[i-k \dots i] = P[0 \dots k]\right\}$

Summary: Knuth-Morris-Pratt Algorithm

Knuth-Morris-Pratt Algorithm

- lps-function gives a succinct representation of δ.
- Using lps to evaluate δ takes **amortized** constant time.
- Constructing lps-table works similar and takes $O(m)$ time.
- KMP algorithm has optimal running time of $O(m + n)$.

Historical Note

In the original paper (1977), the algorithm is not presented in terms of DFAs.
However, the authors point out that *"it was still legitimate to conclude that automata theory had actually been helpful in this practical problem."*

Summary

- Today's topic: **Exact Pattern Matching** (for single patterns without index)
- Reminder: NFAs and DFAs
- Avoid reading text characters more than once
  $\rightarrow$ **NFA-based pattern matching**
- Efficient bit-parallel implementation of pattern matching NFA
  $\rightarrow$ **Shift-And algorithm**
- Best asymptotic worst-case time:
  $\rightarrow$ **DFA-based algorithms**
- Compact representation of DFA transitions:
  $\rightarrow$ **Knuth-Morris-Pratt algorithm**

Possible exam questions

- How can finite automata be used to solve the pattern matching problem?
- What is the difference between NFAs and DFAs?
- What running times can be achieved in NFA/DFA based pattern matching?
- How is the set of active NFA states related to the read text so far?
- Give the formal definition of a pattern matching NFA and explain it.
- Explain the Shift-And algorithm.
- Explain the subset construction (from NFA to DFA).
- Why do the special NFAs studied here have the same number of states as the corresponding DFAs?
- Explain the Knuth-Morris-Pratt (KMP) algorithm and its relation to DFAs.
- How can one construct the *lps* function and in what time?
- What is the running time of KMP (worst case / best case)?