Horspool Algorithm and Automata-Based Algorithms

Algorithms for Sequence Analysis

Sven Rahmann

Summer 2024

Review

Exact Pattern Search (Matching) Problem

Given

finite alphabet Σ, text $T \in \Sigma^n$, pattern $P \in \Sigma^m$; usually $m \ll n$.

Exact Pattern Search (Matching) Problem

Given

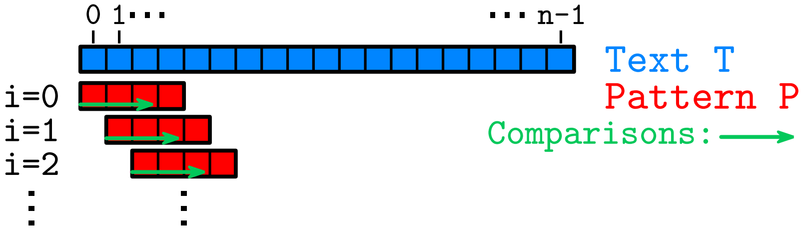
finite alphabet Σ, text $T \in \Sigma^n$, pattern $P \in \Sigma^m$; usually $m \ll n$.

Sought (three variants)

1. **Decision:** Is P a substring of T?
   $\rightsquigarrow$ Is there an $i \in \mathbb{N}$ such that $P = T[i \dots i + m - 1]$?
2. **Counting:** How often does P occur in T?
   $\rightsquigarrow$ Let $M := \{i \in \mathbb{N} \,|\, P = T[i \dots i + m - 1]\}$. Report $|M|$.
3. **Enumeration:** At what positions does P occur in T?
   $\rightsquigarrow$ Report the full set M of match positions.

Naive Pattern Search Algorithm

```
def naive_pattern_search(P, T):
    m, n = len(P), len(T)
    for i in range(n - m + 1):
        if T[i:i+m] == P:  # implicit loop of size m
            yield i
```



Running Time of Naive Search & Possible Improvements

Theorem: Expected Running Time

Let Σ be an alphabet with $|\Sigma| \geq 2$.
Randomly (i.i.d.) choose a pattern of length m and a text of length n over Σ.
Then the worst-case running time of the naïve algorithnm is $O(mn)$,
but the expected running time is $O(E_m \cdot n) = O(n)$ with a small constant $E_m < 2$.

Thoughts

1. Can we shift window by more than one character? $\rightarrow$ Horspool algorithm
2. We "touch" the same characters in `T` multiple times.
   Can we "re-use" information from preceding comparisons?
   $\rightarrow$ Automata-based algorithms

The Horspool Algorithm

Motivation: Horspool Algorithm

Question

When and how can the window be shifted by more than one position?

Ideas

- Compare pattern **right-to-left** to text window.
- Characters not occurring in pattern $\rightarrow$ large shift

(Extreme) Example

```
AAAAAAAAAAAAAAAAAAAAAAAAAA
BBBBBB
```

Motivation: Horspool Algorithm

Question

When and how can the window be shifted by more than one position?

Ideas

- Compare pattern **right-to-left** to text window.
- Characters not occurring in pattern $\rightarrow$ large shift

(Extreme) Example

```
AAAAAAAAAAAAAAAAAAAAAAAAAAA
      BBBBBB
```

Motivation: Horspool Algorithm

Question

When and how can the window be shifted by more than one position?

Ideas

- Compare pattern **right-to-left** to text window.
- Characters not occurring in pattern $\rightarrow$ large shift

(Extreme) Example

```
AAAAAAAAAAAAAAAAAAAAAAAAAAA
           BBBBBB
```

Motivation: Horspool Algorithm

Question

When and how can the window be shifted by more than one position?

Ideas

- Compare pattern **right-to-left** to text window.
- Characters not occurring in pattern $\rightarrow$ large shift

(Extreme) Example

```
AAAAAAAAAAAAAAAAAAAAAAAAAAA
                  BBBBBB
```

Motivation: Horspool Algorithm

Question

When and how can the window be shifted by more than one position?

Ideas

- Compare pattern **right-to-left** to text window.
- Characters not occurring in pattern $\rightarrow$ large shift

(Extreme) Example

```
AAAAAAAAAAAAAAAAAAAAAAAAAAA
                   BBBBBB
```

Best-case time $O(n/m)$, but worst-case time can be $O(nm)$.

Horspool Algorithm

Approach

- Window-based pattern search algorithm
- Shift determined by **last character** in window

Example

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

Question: How far can we shift the window without missing pattern occurrences?

Text: $\underbrace{\texttt{?????A}}\texttt{?????}$ $\underbrace{\texttt{?????B}}\texttt{?????}$ $\underbrace{\texttt{?????C}}\texttt{??????}$

Horspool Algorithm

Approach

- Window-based pattern search algorithm
- Shift determined by **last character** in window

Example

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

Question: How far can we shift the window without missing pattern occurrences?

Text:	`?????A?????`	`?????B?????`	`?????C??????`
Pattern:	`BAAAAB`	`BAAAAB`	`BAAAAB`

Horspool Algorithm

Approach

- Window-based pattern search algorithm
- Shift determined by **last character** in window

Example

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

Question: How far can we shift the window without missing pattern occurrences?

Text: `?????A?????` `?????B?????` `?????C??????`

Pattern: `BAAAAB` `BAAAAB` `BAAAAB`

In the next step, the currently last text window character must align with the rightmost equal character in the pattern (w/o the last one).

Horspool Algorithm (formal)

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

Text:	?????A?????	?????B?????	?????C??????
Pattern:	BAAAAB	BAAAAB	BAAAAB

```python
def horspool_preprocessing(Sigma, P):
    shifts = dict()
    for c in Sigma:
        shifts[c] = len(P)
    for i in range(len(P)-1):
        shifts[P[i]] = len(P) - i - 1
    return shifts
```

Horspool Algorithm (formal)

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

Text:	?????A?????	?????B?????	?????C??????
Pattern:	BAAAAB	BAAAAB	BAAAAB

```python
def horspool_preprocessing(Sigma, P):
    shifts = dict()
    for c in Sigma:
        shifts[c] = len(P)
    for i in range(len(P)-1):
        shifts[P[i]] = len(P) - i - 1
    return shifts
```

shifts:

A	B	C
6	6	6

Horspool Algorithm (formal)

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

Text:	?????A?????	?????B?????	?????C??????
Pattern:	BAAAAB	BAAAAB	BAAAAB

```
def horspool_preprocessing(Sigma, P):
    shifts = dict()
    for c in Sigma:
        shifts[c] = len(P)
    for i in range(len(P)-1):
        shifts[P[i]] = len(P) - i - 1
    return shifts
```

shifts:

A	B	C
6	5	6

$i = 0$ $\quad p =$ BAAAAB

Horspool Algorithm (formal)

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

Text:	?????A?????	?????B?????	?????C??????
Pattern:	BAAAAB	BAAAAB	BAAAAB

```python
def horspool_preprocessing(Sigma, P):
    shifts = dict()
    for c in Sigma:
        shifts[c] = len(P)
    for i in range(len(P)-1):
        shifts[P[i]] = len(P) - i - 1
    return shifts
```

shifts:

A	B	C
4	5	6

$i = 1$ $\quad p = \texttt{BAAAAB}$

Horspool Algorithm (formal)

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

Text:	`?????A?????`	`?????B?????`	`?????C??????`
Pattern:	`BAAAAB`	`BAAAAB`	`BAAAAB`

```python
def horspool_preprocessing(Sigma, P):
    shifts = dict()
    for c in Sigma:
        shifts[c] = len(P)
    for i in range(len(P)-1):
        shifts[P[i]] = len(P) - i - 1
    return shifts
```

shifts:

A	B	C
3	5	6

$i = 2$ $\quad p = \texttt{BAAAAB}$

Horspool Algorithm (formal)

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

Text:	?????A?????	?????B?????	?????C??????
Pattern:	BAAAAB	BAAAAB	BAAAAB

```
def horspool_preprocessing(Sigma, P):
    shifts = dict()
    for c in Sigma:
        shifts[c] = len(P)
    for i in range(len(P)-1):
        shifts[P[i]] = len(P) - i - 1
    return shifts
```

shifts:

A	B	C
2	5	6

$i = 3$ p = BAAAAB

Horspool Algorithm (formal)

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

Text:	`?????A?????`	`?????B?????`	`?????C??????`
Pattern:	`BAAAAB`	`BAAAAB`	`BAAAAB`

```
def horspool_preprocessing(Sigma, P):
    shifts = dict()
    for c in Sigma:
        shifts[c] = len(P)
    for i in range(len(P)-1):
        shifts[P[i]] = len(P) - i - 1
    return shifts
```

shifts:

A	B	C
1	5	6

$i = 4$ $\quad p =$ BAAAAB

Horspool-Algorithm (formal) II

```
def horspool_search(sigma, P, T):
    shifts = horspool_preprocessing(sigma, P)
    i = len(P) - 1
    while i < len(T):
        if T[i:i-len(P):-1] == P[::-1]:  # implicit loop
            yield i  # end index of match
        i += shifts[T[i]]
```

Text: ABBCACBABAABBAAAABAABCAC...

BAAAAB

shifts:

A	B	C
1	5	6

Horspool-Algorithm (formal) II

```
def horspool_search(sigma, P, T):
    shifts = horspool_preprocessing(sigma, P)
    i = len(P) - 1
    while i < len(T):
        if T[i:i-len(P):-1] == P[::-1]:  # implicit loop
            yield i  # end index of match
        i += shifts[T[i]]
```

Text: ABBCACBABAABBAAAABAABCAC...

BAAAAB

shifts:

A	B	C
1	5	6

Horspool-Algorithm (formal) II

```
def horspool_search(sigma, P, T):
    shifts = horspool_preprocessing(sigma, P)
    i = len(P) - 1
    while i < len(T):
        if T[i:i-len(P):-1] == P[::-1]:  # implicit loop
            yield i  # end index of match
        i += shifts[T[i]]
```

Text: ABBCACBABAABBAAAABAABCAC...

BAAAAB

shifts:

A	B	C
1	5	6

Horspool-Algorithm (formal) II

```
def horspool_search(sigma, P, T):
    shifts = horspool_preprocessing(sigma, P)
    i = len(P) - 1
    while i < len(T):
        if T[i:i-len(P):-1] == P[::-1]:  # implicit loop
            yield i  # end index of match
        i += shifts[T[i]]
```

Text: ABBCACBABAABBAAAABAABCAC...

$\underbrace{\texttt{BAAAAB}}$

BAAAAB

shifts:

A	B	C
1	5	6

Horspool-Algorithm (formal) II

```
def horspool_search(sigma, P, T):
    shifts = horspool_preprocessing(sigma, P)
    i = len(P) - 1
    while i < len(T):
        if T[i:i-len(P):-1] == P[::-1]:  # implicit loop
            yield i  # end index of match
        i += shifts[T[i]]
```

Text: ABBCACBABAABBAAAABAABCAC...

BAAAAB

shifts:

A	B	C
1	5	6

fast for large alphabets and long patterns

Automata-Based Algorithms

Deterministic Finite Automaton (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**.

Deterministic Finite Automaton (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**.

Example

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 Automaton (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**.

Non-Deterministic Finite Automaton (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**.

Example

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

Extending the Transition Function

- Original NFA transition function: $\Delta\colon Q \times \Sigma \to 2^Q$
- For notational convenience, we extend it in two ways.

Extending the Transition Function

- Original NFA transition function: $\Delta\colon Q \times \Sigma \to 2^Q$
- For notational convenience, we extend it in two ways.

Extension to sets of states

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

Extending the Transition Function

- Original NFA transition function: $\Delta\colon Q \times \Sigma \to 2^Q$
- For notational convenience, we extend it in two ways.

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 $c \in \Sigma$.

NFA to Solve the Pattern Search Problem

Goal

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

NFA to Solve the Pattern Search Problem

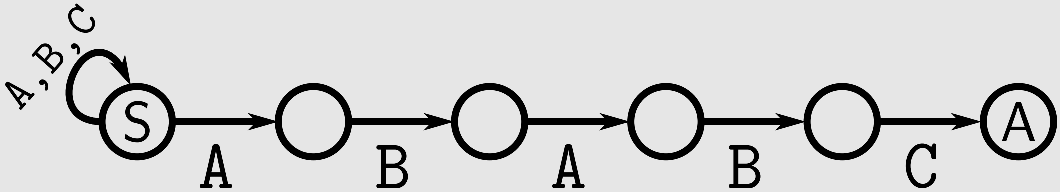
Goal

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

Approach

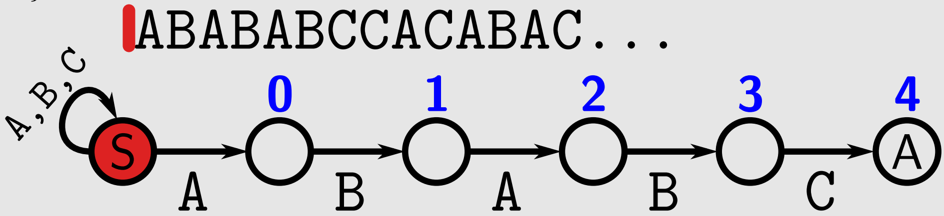
- "Linear chain" of states
- Start state remains always active

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



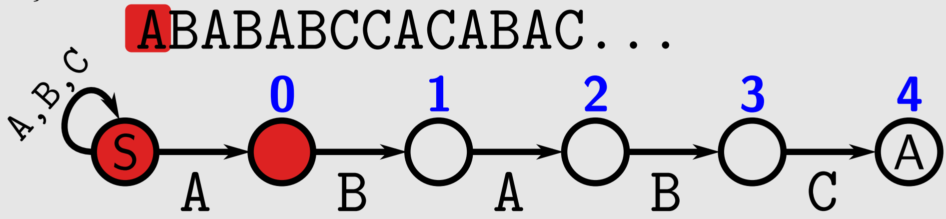
Using the NFA for Pattern Search

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



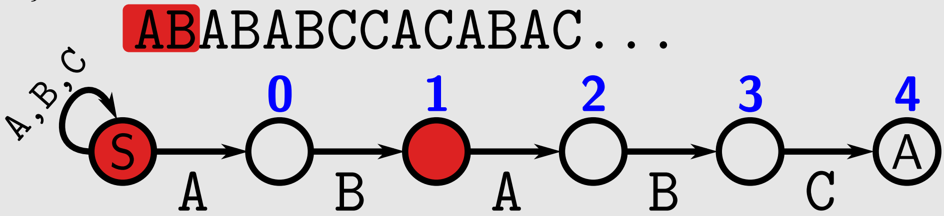
Using the NFA for Pattern Search

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



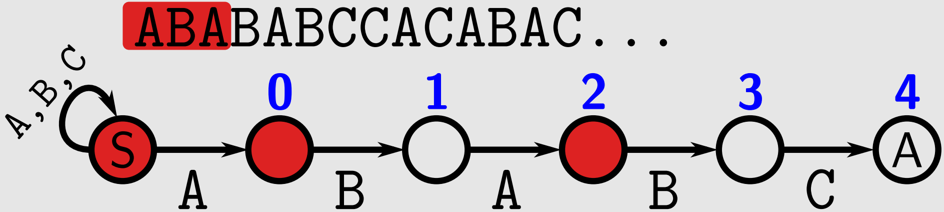
Using the NFA for Pattern Search

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



Using the NFA for Pattern Search

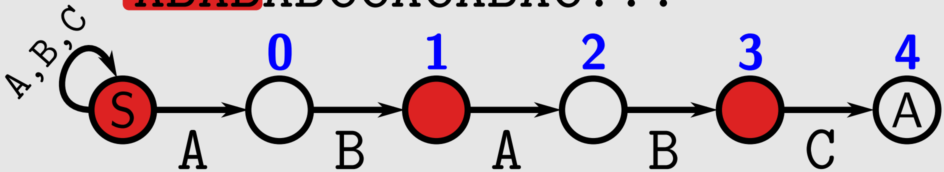
$\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$ and $P = \mathtt{ABABC}$



Using the NFA for Pattern Search

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

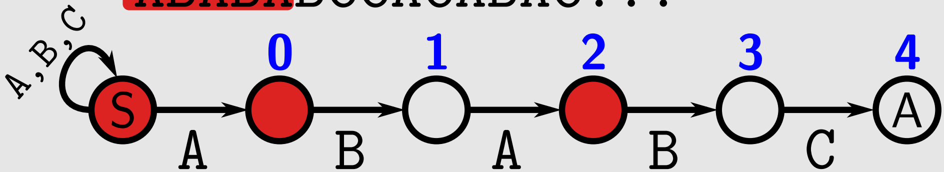
ABABABCCACABAC...



Using the NFA for Pattern Search

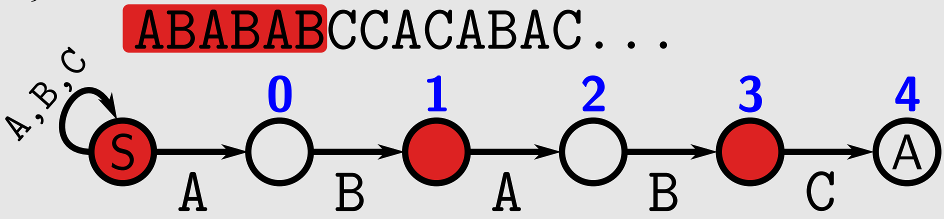
$\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$ and $P = \mathtt{ABABC}$

ABABABCCACABAC...



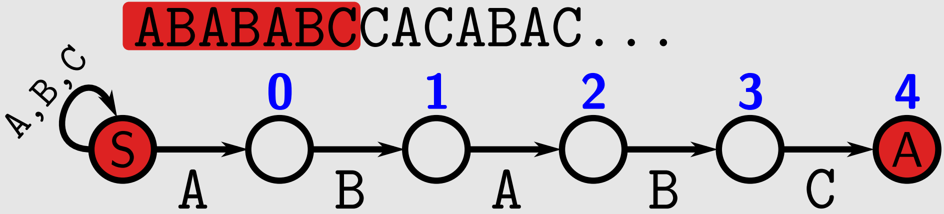
Using the NFA for Pattern Search

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



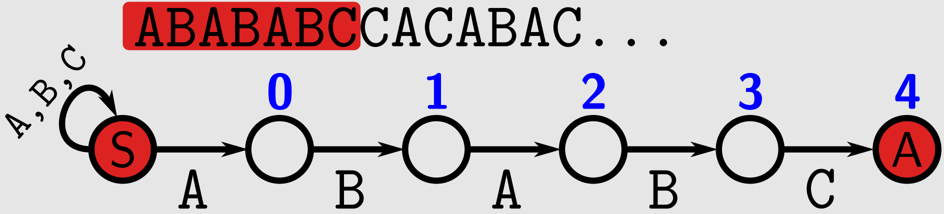
Using the NFA for Pattern Search

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



Using the NFA for Pattern Search

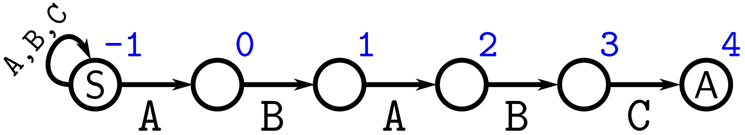
$\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$ and $P = \mathtt{ABABC}$



Things left to do:

- Formally define this automaton.
- Give an efficient implementation.

Pattern Search NFA (Formal)



Pattern Search NFA for pattern $P \in \Sigma^m$

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

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

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

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

Correctness

Lemma: NFA state set invariant

Let $A \subset Q$ be a set of active states of the NFA. Then, $q \in A \setminus \{-1\}$ iff the last $q+1$ read characters equal the pattern's 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.

Correctness

Lemma: NFA state set invariant

Let $A \subset Q$ be a set of active states of the 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.

Proof: Follows directly from the NFA definition.

Theorem: Correctness of Pattern Search NFA

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

Proof: Follows immediately from the above lemma.

Running Time

Derivation

- In an NFA, more than one state can be active; $m+1 = O(m)$ states.
- In the Pattern Search NFA, each target set's size is bounded by $2 = O(1)$.
- Thus, each step (text character) takes $O(m)$ time worst-case.
- Total: $O(mn)$, same as naive algorithm.

Running Time

Derivation

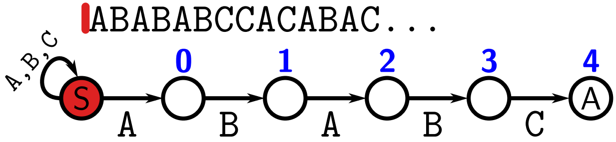
- In an NFA, more than one state can be active; $m + 1 = O(m)$ states.
- In the Pattern Search NFA, each target set's size is bounded by $2 = O(1)$.
- Thus, each step (text character) takes $O(m)$ time worst-case.
- Total: $O(mn)$, same as naive algorithm.

Improvement: Bit Parallelism

- On modern CPUs, logical and arithmetic operations on many bits (64 bits) take place in parallel in constant time: "**Bit parallelism**" $(+, -, \cdot, /, \oplus, \&, |, \sim, \ll, \gg)$
- The Pattern Search NFA is a linear chain of states, like bits in a CPU register.
- We only need one bit to represent whether a state is active or not.
- **Note:** States are numbered from left to right, bits from right to left!

Efficient Implementation: The Shift-And Algorithm

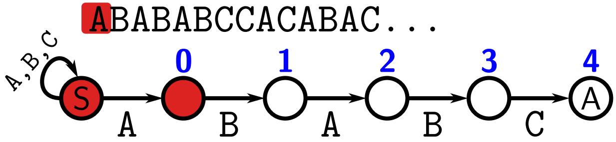
- We encode active states (without start) as a bit vector D, initially $D = 0$.
- The pattern is encoded in bit masks, one for each character.
- **Update:** $D \leftarrow ((D \ll 1) \mid 1)\ \&\ \mathrm{mask}(\alpha)$, where α is the current text character:
  shift $\ll 1$: propagates activity to next state, $\mid 1$ propagates the start state;
  and $\&\ \mathrm{mask}(\alpha)$: removes falsely propagated activity.
- After each update, test whether accepting state $m-1$ is active.



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

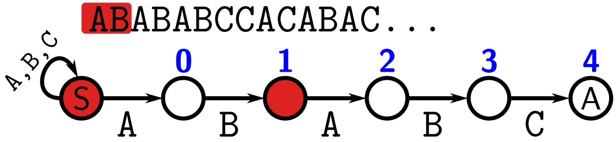
- We encode active states (without start) as a bit vector D, initially $D = 0$.
- The pattern is encoded in bit masks, one for each character.
- **Update:** $D \leftarrow ((D \ll 1) \mid 1)\ \&\ \text{mask}(\alpha)$, where α is the current text character:
  shift $\ll 1$: propagates activity to next state, $\mid 1$ propagates the start state;
  and $\&\ \text{mask}(\alpha)$: removes falsely propagated activity.
- After each update, test whether accepting state $m - 1$ is active.



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

- We encode active states (without start) as a bit vector D, initially $D = 0$.
- The pattern is encoded in bit masks, one for each character.
- **Update:** $D \leftarrow ((D \ll 1) \mid 1)\ \&\ \text{mask}(\alpha)$, where α is the current text character:
  shift $\ll 1$: propagates activity to next state, $\mid 1$ propagates the start state;
  and $\&\ \text{mask}(\alpha)$: removes falsely propagated activity.
- After each update, test whether accepting state $m - 1$ is active.

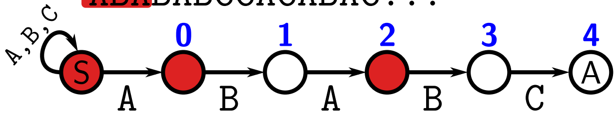


	0	1	2	3	4
D:	0	1	0	0	0
mask(A):	1	0	1	0	0
mask(B):	0	1	0	1	0
mask(C):	0	0	0	0	1

Efficient Implementation: The Shift-And Algorithm

- We encode active states (without start) as a bit vector D, initially $D = 0$.
- The pattern is encoded in bit masks, one for each character.
- **Update:** $D \leftarrow ((D \ll 1) \mid 1)\ \&\ \text{mask}(\alpha)$, where α is the current text character:
  shift $\ll 1$: propagates activity to next state, $\mid 1$ propagates the start state;
  and $\&\ \text{mask}(\alpha)$: removes falsely propagated activity.
- After each update, test whether accepting state $m - 1$ is active.

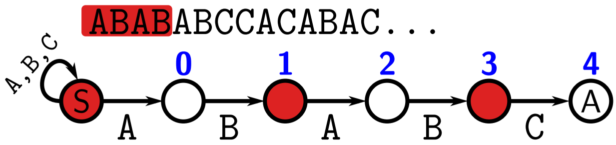
ABABABCCACABAC...



	0	1	2	3	4
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

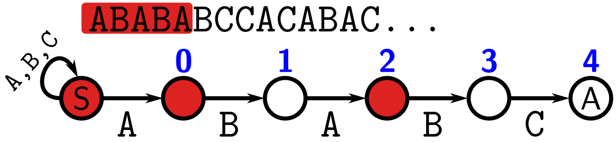
- We encode active states (without start) as a bit vector D, initially $D = 0$.
- The pattern is encoded in bit masks, one for each character.
- **Update:** $D \leftarrow ((D \ll 1) \mid 1)$ & $\mathrm{mask}(\alpha)$, where α is the current text character:
  shift $\ll 1$: propagates activity to next state, $\mid 1$ propagates the start state;
  and & $\mathrm{mask}(\alpha)$: removes falsely propagated activity.
- After each update, test whether accepting state $m-1$ is active.



	0	1	2	3	4
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

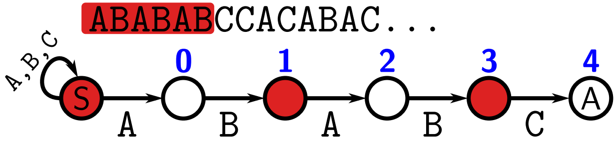
- We encode active states (without start) as a bit vector D, initially $D = 0$.
- The pattern is encoded in bit masks, one for each character.
- **Update:** $D \leftarrow ((D \ll 1) \mid 1)$ & mask(α), where α is the current text character:
  shift $\ll 1$: propagates activity to next state, $\mid 1$ propagates the start state;
  and & mask(α): removes falsely propagated activity.
- After each update, test whether accepting state $m - 1$ is active.



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

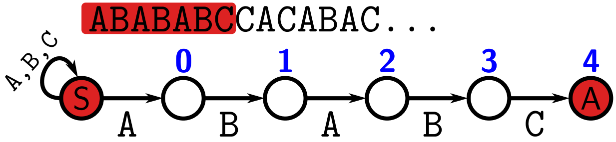
- We encode active states (without start) as a bit vector D, initially $D = 0$.
- The pattern is encoded in bit masks, one for each character.
- **Update:** $D \leftarrow ((D \ll 1) \mid 1)\ \&\ \text{mask}(\alpha)$, where α is the current text character:
  shift $\ll 1$: propagates activity to next state, $\mid 1$ propagates the start state;
  and $\&$ mask(α): removes falsely propagated activity.
- After each update, test whether accepting state $m - 1$ is active.



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

- We encode active states (without start) as a bit vector D, initially $D = 0$.
- The pattern is encoded in bit masks, one for each character.
- **Update:** $D \leftarrow ((D \ll 1) \mid 1)$ & $\mathrm{mask}(\alpha)$, where α is the current text character:
  shift $\ll 1$: propagates activity to next state, $\mid 1$ propagates the start state;
  and & $\mathrm{mask}(\alpha)$: removes falsely propagated activity.
- After each update, test whether accepting state $m-1$ is active.



	0	1	2	3	4
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

Code for Shift-And Algorithm

```python
from collections import defaultdict

def shift_and(P, T):
    masks = defaultdict(int)  # masks[c] == 0 if c not in masks
    bit = 1
    for c in P:
        masks[c] |= bit
        bit *= 2
    accept_state = bit // 2
    D = 0  # bit-mask of active states
    for i, c in enumerate(T):
        D = ((D << 1) | 1) & masks[c]
        if (D & accept_state):
            yield i
```

Running Time of Shift-And Algorithm

- m: Pattern length
- n: Text length
- w: Machine register width (constant, 64)

Running time

If $m \leq w$, then the shift-and algorithm runs in $O(m + n)$ time.
Generally (i.e., when m 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.

Idea for Improvement: Shift-Or

Shift-And: 3 operations per character

$D \leftarrow ((D \ll 1) \mid 1) \;\&\; \mathsf{mask}(\alpha)$

the $\mid 1$ propagates the (always active) start state.

Idea for Improvement: Shift-Or

Shift-And: 3 operations per character

$D \leftarrow ((D \ll 1) \mid 1) \;\&\; \mathsf{mask}(\alpha)$

the $\mid 1$ propagates the (always active) start state.

Idea for a faster variant

- We can save the $\mid 1$ operation if we **invert the bit logic**.
- By a one-bit shift ($\ll 1$), a 0-bit is shifted in automatically.
- So, let 0 mean "active", let 1 mean "inactive".
- Use a type of fixed bit width (like `numpy.uint64`), not Python `int`.
- Update becomes $D \leftarrow (D \ll 1) \mid \mathsf{imask}(\alpha)$: **Shift-Or**
- Need to use inverted bit masks; invert acceptance test.
- Still need to start with all bits inactive (now all 1-bits, e.g. `uint64(-1)`).

Summary

- Idea and Motivation of **Horspool's Algorithm**
- Details: preprocessing and shifting
- Running time: best-case $O(n/m)$ vs. worst-case $O(nm)$

- Review of automata: DFAs and NFAs
- Pattern Search NFA, informally and formally
- Bit-parallel implementation: **Shift-And Algorithm**
- Running time analysis: $O(nm/w)$ with register width w
- Use case is for $m/w \leq 1$, i.e., $m \leq w = 64$.
- Small improvement: **Shift-Or Algorithm** (inverted bit logic)

Possible exam questions

- Run Horspool's Algorithm (preprocessing + search) on an example.
- Explain how to construct the shift table for Horspool's Algorithm.
- Give non-trivial examples with best-case and worst-case running times for Horspool's Algorithm.
- For which pattern properties is Horspool's Algorithm fast or slow?
- Construct an NFA (DFA) that accepts a given set of strings.
- Where is the "non-determinism" in NFAs (as opposed to DFAs)?
- Construct the Pattern Search NFA for a given pattern.
- Formally explain the Pattern Search NFA construction.
- Characterize the set of active NFA states in a Pattern Search NFA after reading a text character.
- Explain the idea of the Shift-And Algorithm.
- Run the Shift-And Algorithm on an example.
- Explain the idea of the Shift-Or Algorithm. Why is it faster than Shift-And?