# Pattern Matching

# Pattern Matching

- Given:
  - a word $w$ (the text) of length $n$, and
  - a regular expression p (the pattern) of length $m$

  determine
  - the smallest number $k'$ such that some $[k, k']$-factor of $w$ belongs to $L(p)$.

# NFA-based solution

*PatternMatchingNFA*$(t, p)$

**Input:** text $t = a_1 \ldots a_n \in \Sigma^+$, pattern $p \in \Sigma^*$

**Output:** the first occurrence of $p$ in $t$, or $\perp$ if no such occurrence exists.

```
1  A ← RegtoNFA(Σ* p)
2  S ← {q_0}
3  for all k = 0 to n − 1 do
4      if S ∩ F ≠ ∅ then return k
5      S ← δ(S, a_{k+1})
6  return ⊥
```

- Line 1 takes $O(m^3)$ time, output has $O(m)$ states
- Loop is executed at most $n$ times
- One iteration takes $O(s^2)$ time , where $s$ is the number of states of $A$
- Since $s = O(m)$, the total runtime is $O(m^3 + nm^2)$ , and $O(nm^2)$ for $m \leq n$ .

# DFA-based solution

*PatternMatchingDFA*$(t, p)$
**Input:** text $t = a_1 \ldots a_n \in \Sigma^+$, pattern $p$
**Output:** the first occurrence of $p$ in $t$, or $\perp$ if no such occurrence exists.

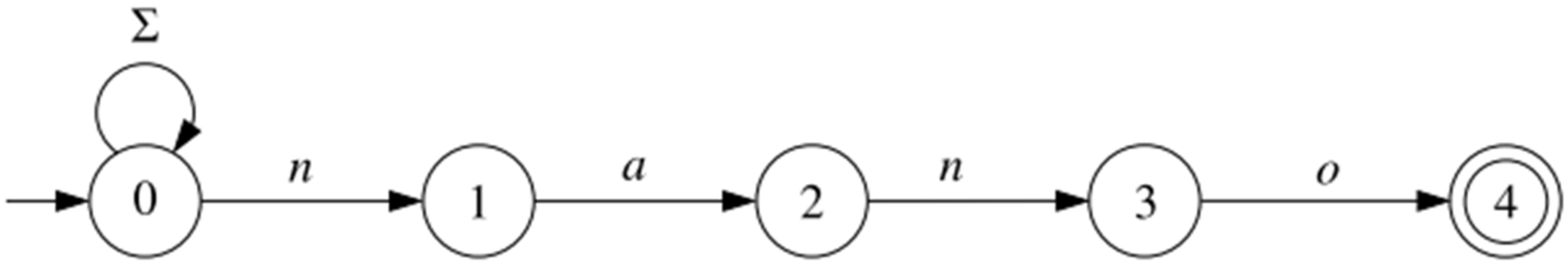
```
1  A ← NFAtoDFA(RegtoNFA(Σ* p))
2  q ← q_0
3  for all k = 0 to n − 1 do
4      if q ∈ F then return k
5      q ← δ(q, a_{k+1})
6  return ⊥
```

- Line 1 takes $2^{O(m)}$ time
- Loop is executed at most $n$ times
- One iteration takes constant time
- Total runtime is $O(n) + 2^{O(m)}$

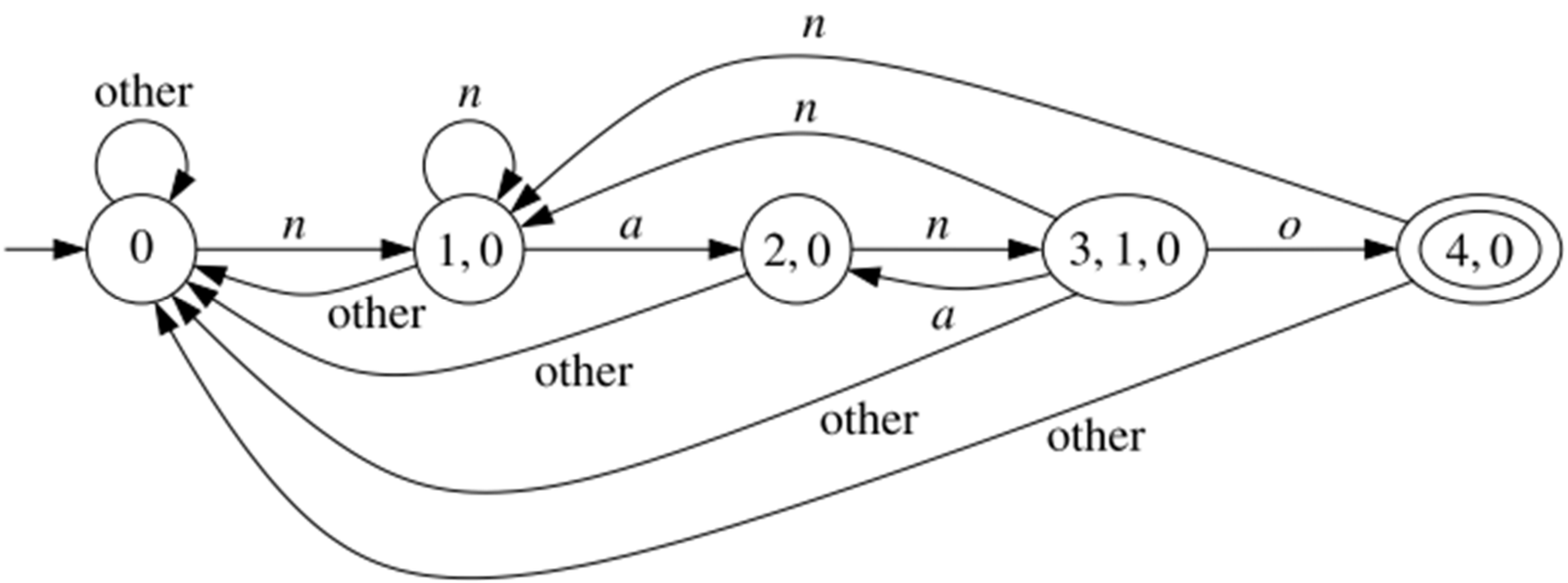
# The word case

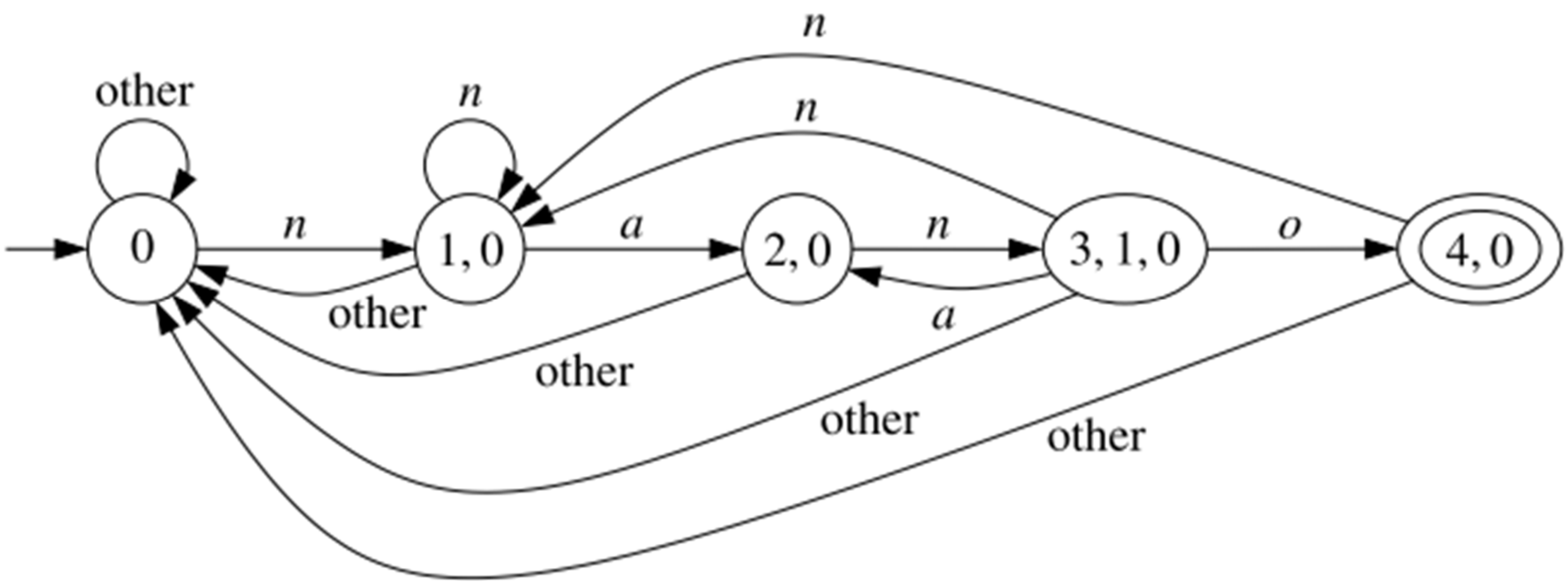
- The pattern $p$ is a word of length $m$
- Naive algorithm: move a window of size m along the word one letter at a time, and compare with p after each step. Runtime: $O(nm)$
- We give an algorithm with $O(n+m)$ runtime for any alphabet of size $0 \leq |\Sigma| \leq n$ .
- First we explore in detail the shape of the DFA for $\Sigma^* p$ .

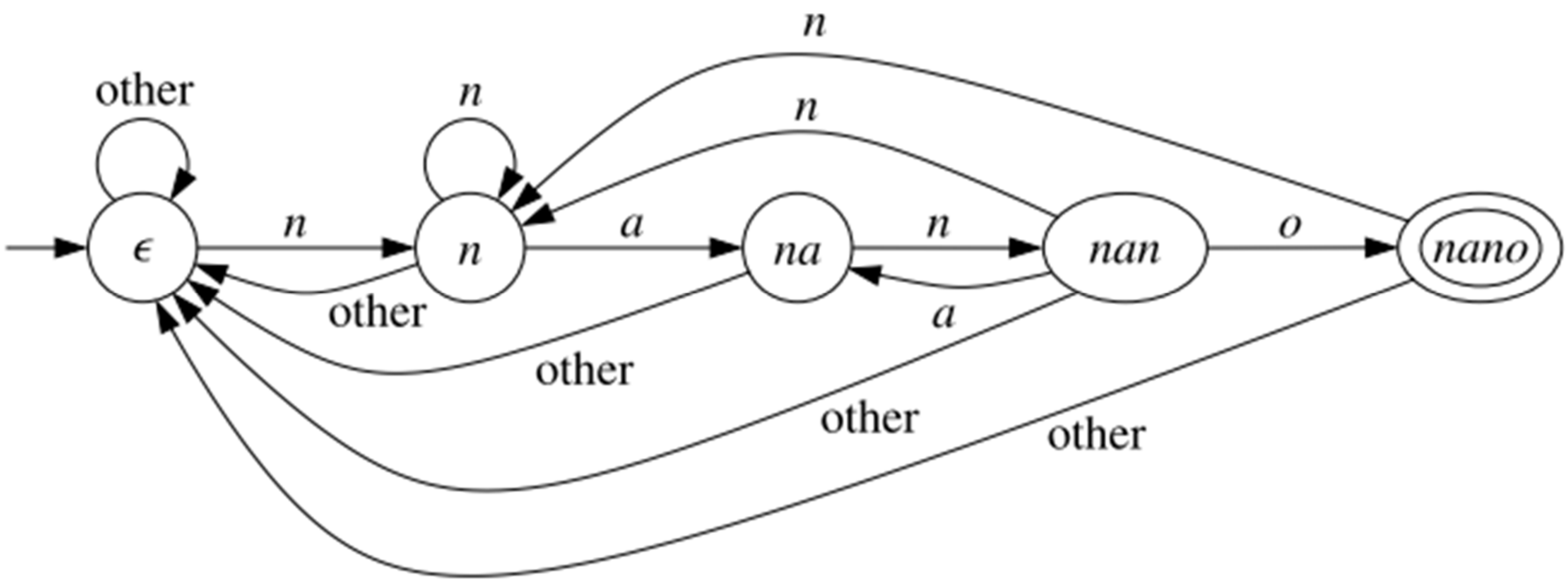
# Obvious NFA for $\Sigma^* p$ and $p = nano$



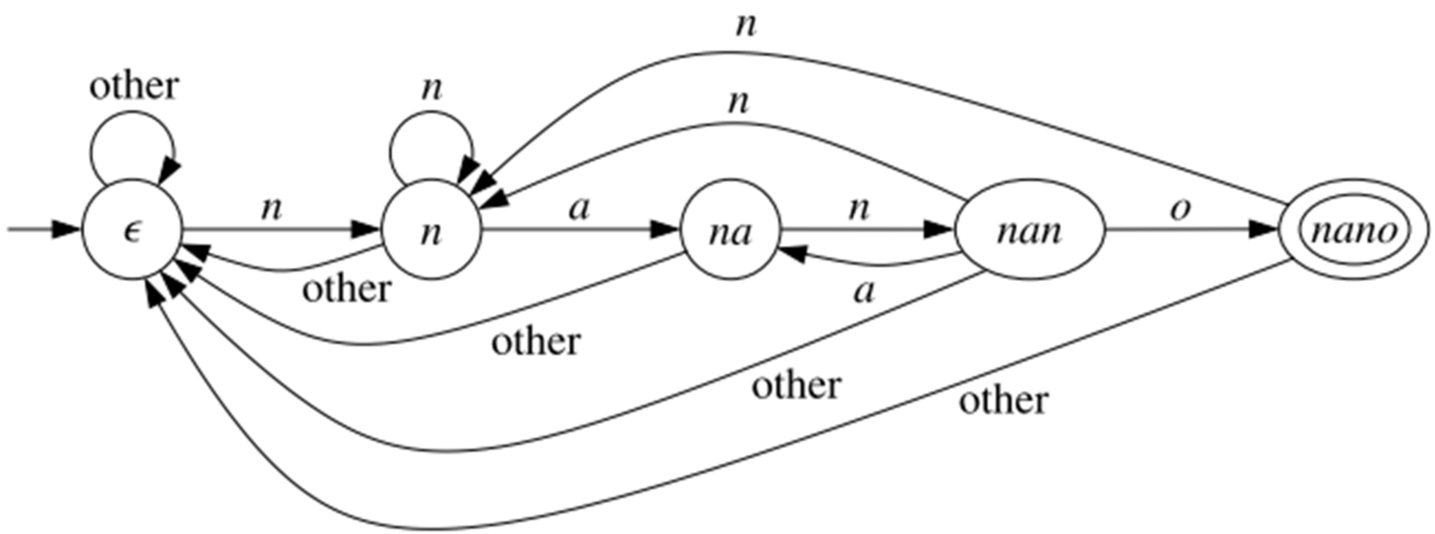

# Result of applying NFAtoDFA




other
n
n
n
0
n
1, 0
a
2, 0
n
3, 1, 0
o
4, 0
other
a
other
other
other

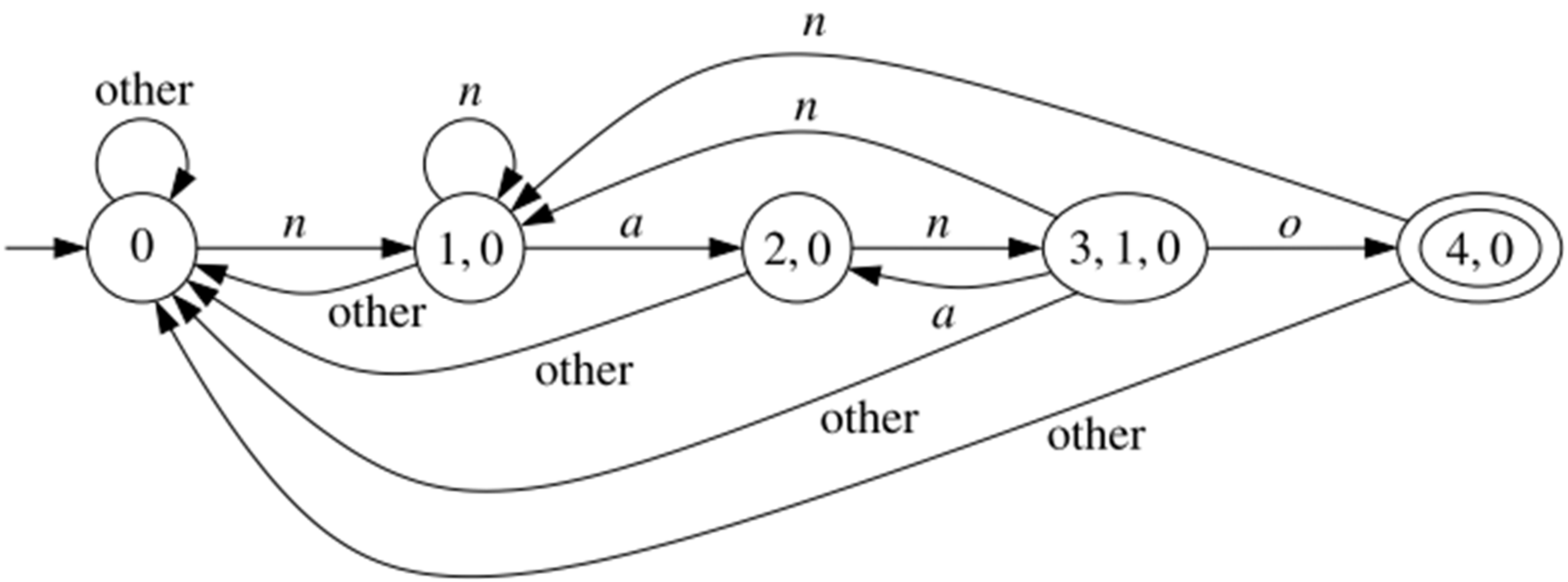

other
n
n
n
$\epsilon$
n
n
a
na
n
nan
o
nano
other
a
other
other
other

# Intuition



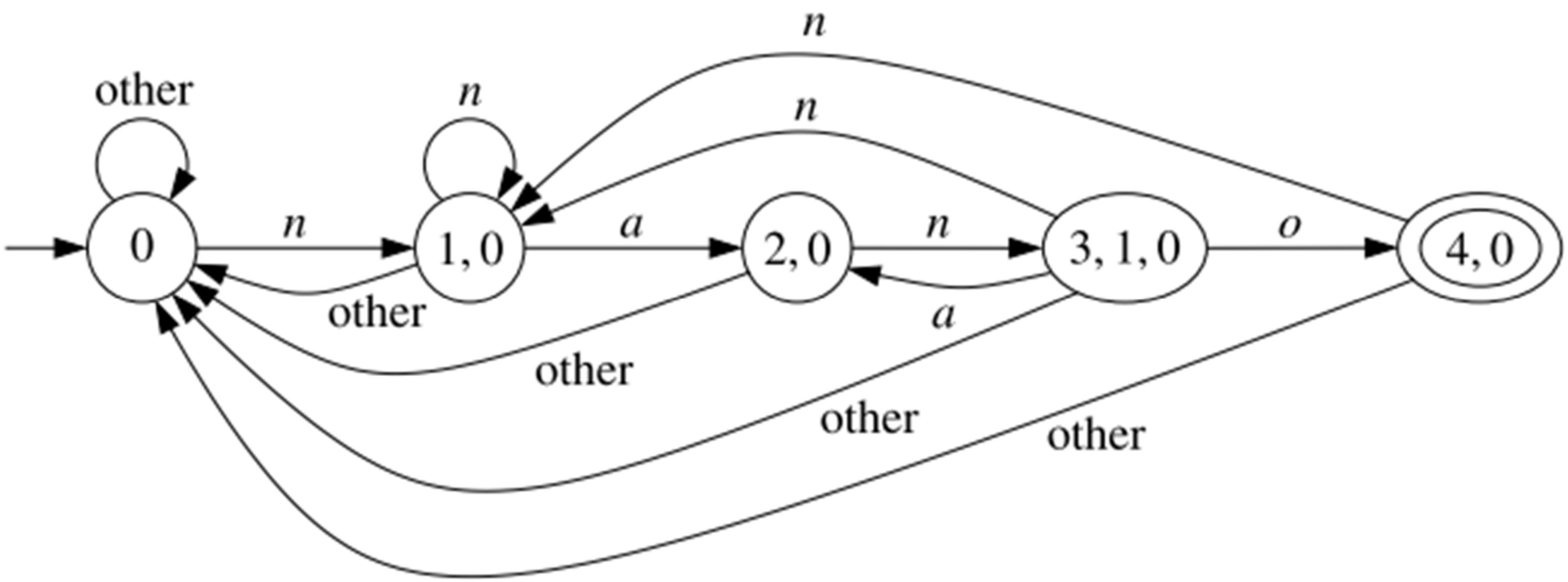

- Transitions of the „spine“ correspond to hits: the next letter is the one that „makes progress“ towards nano
- Other transitions correspond to misses, i.e., „wrong letters“ and „throw the automaton back“

# Observations




- For every state $i = 0,1,\ldots,4$ of the NFA there is exactly one state $S$ of the DFA such that i is the largest state of $S$.
- For every state $S$ of the DFA, with the exception of $S = \{0\}$, the result of removing the largest state is again a state of the DFA.

# Observations




- For every state $i = 0,1, \ldots, 4$ of the NFA there is exactly one state $S$ of the DFA such that i is the largest state of $S$.
- For every state $S$ of the DFA, with the exception of $S = \{0\}$, the result of removing the largest state is again a state of the DFA.
- Do these properties hold for every pattern $p$?

# Heads and tails, hits and misses

- Head of $S$, denoted $h(S)$ : largest state of $S$
- Tail of $S$, denoted $t(S)$ : rest of the state
- Example: $h(\{3,1,0\}) = 3, \; t(\{3,1,0\}) = \{1,0\}$

- Given a state $S$, the letter leading to the next state in the „spine“ is the (unique) hit letter for $S$
- All other letters are miss letters for $S$
- Example: hit for {3,1,0} is *o*, whereas *n* or *a* are misses

- Fund. Prop: Let $S_k$ be the $k$-th state picked from the worklist during the execution of *NFAtoDFA($A_p$)*.
  (1) $h(S_k) = k$,
  (2) If $k > 0$, then $t(S_k) = S_l$ for some $l < k$

Proof Idea:

- (1) and (2) hold for $S_0 = \{0\}$.
- For $S_k$ we look at $\delta(S_k, a)$ for each $a$, where $\delta$ transition relation of $A_p$ .
- By i.h. we have $S_k = \{k\} \cup S_l$ for some $l < k$
- We distinguish two cases: *a* is a hit for $S_k$, and *a* is a miss for $S_k$ .

- $S_k = \{k\} \cup S_l$ for some $l < k$
- $\delta(S_k, a) = \delta(k, a) \cup \delta(S_l, a)$

$$\{k\} \quad \cup \quad S_l$$

Hit: $\quad \downarrow a \qquad\qquad \downarrow a$

$$\{k+1\} \cup \delta(S_l, a)$$

- $S_k = \{k\} \cup S_l$ for some $l < k$
- $\delta(S_k, a) = \delta(k, a) \cup \delta(S_l, a)$

Hit:

$$\{k\} \quad \cup \quad S_l$$

$$a \downarrow \qquad\qquad a \downarrow$$

$$\{k+1\} \cup \delta(S_l, a)$$

Added to the worklist earlier, and so some $S_{l'}$

- $S_k = \{k\} \cup S_l$ for some $l < k$
- $\delta(S_k, a) = \delta(k, a) \cup \delta(S_l, a)$

Hit:

$$\begin{array}{ccc} \{k\} & \cup & S_l \\ \downarrow a & & \downarrow a \\ \{k+1\} & \cup & \delta(S_l, a) \\ = & & = \\ \{k+1\} & \cup & S_{l'} \end{array}$$

- $S_k = \{k\} \cup S_l$ for some $l < k$
- $\delta(S_k, a) = \delta(k, a) \cup \delta(S_l, a)$

$$\{k\} \quad \cup \quad S_l$$

Miss: $\quad \Big\downarrow a \qquad\qquad \Big\downarrow a$

$$\emptyset \quad \cup \quad \delta(S_l, a)$$

- $S_k = \{k\} \cup S_l$ for some $l < k$
- $\delta(S_k, a) = \delta(k, a) \cup \delta(S_l, a)$

Miss:

$$\begin{array}{ccc} \{k\} & \cup & S_l \\ a\downarrow & & a\downarrow \\ \emptyset & \cup & \delta(S_l, a) \\ & & = \\ & & S_{l'} \end{array}$$

# Consequences

Prop: The result of applying *NFAtoDFA*$(A_p)$, where $A_p$ is the obvious NFA for $\Sigma^* p$ , yields a minimal DFA with $m$ states and $|\Sigma|m$ transitions.

Proof: All states of the DFA accept different languages.

So: concatenating *NFAtoDFA* *and PatternMatchingDFA* yields a $O(n + |\Sigma|m)$ algorithm.

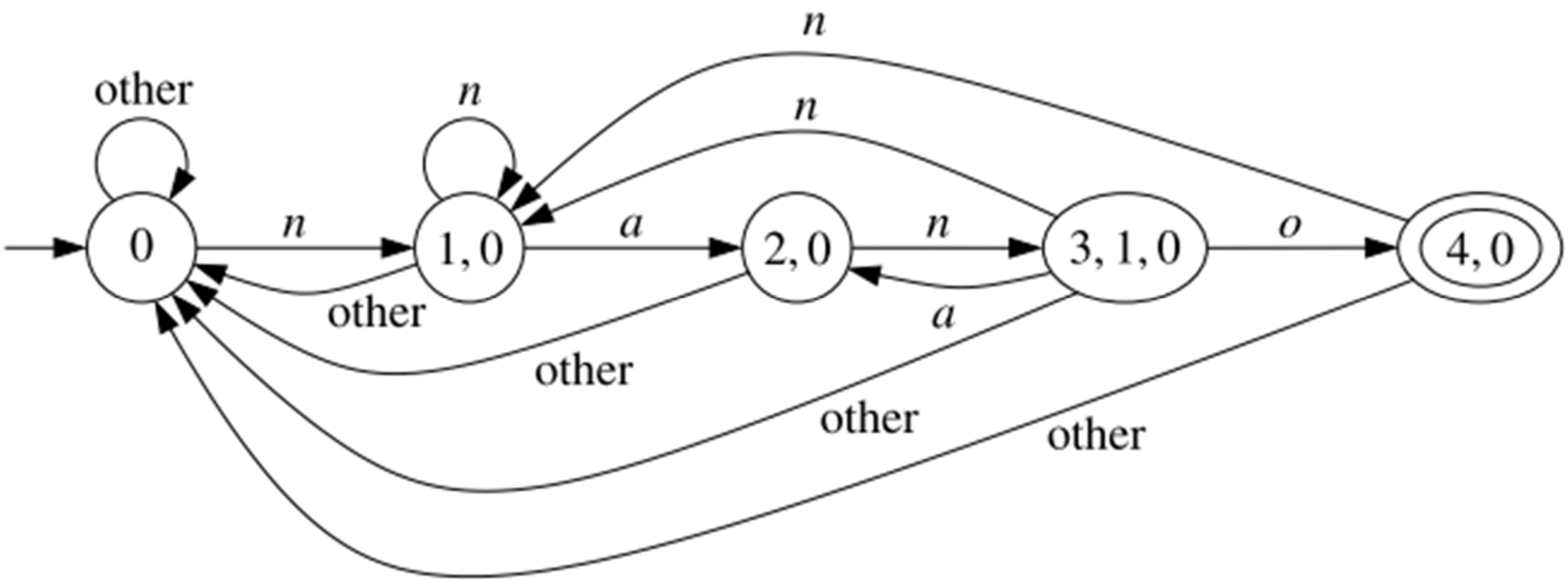
- Good enough for constant alphabet
- Not good enough for $|\Sigma| = O(n)$

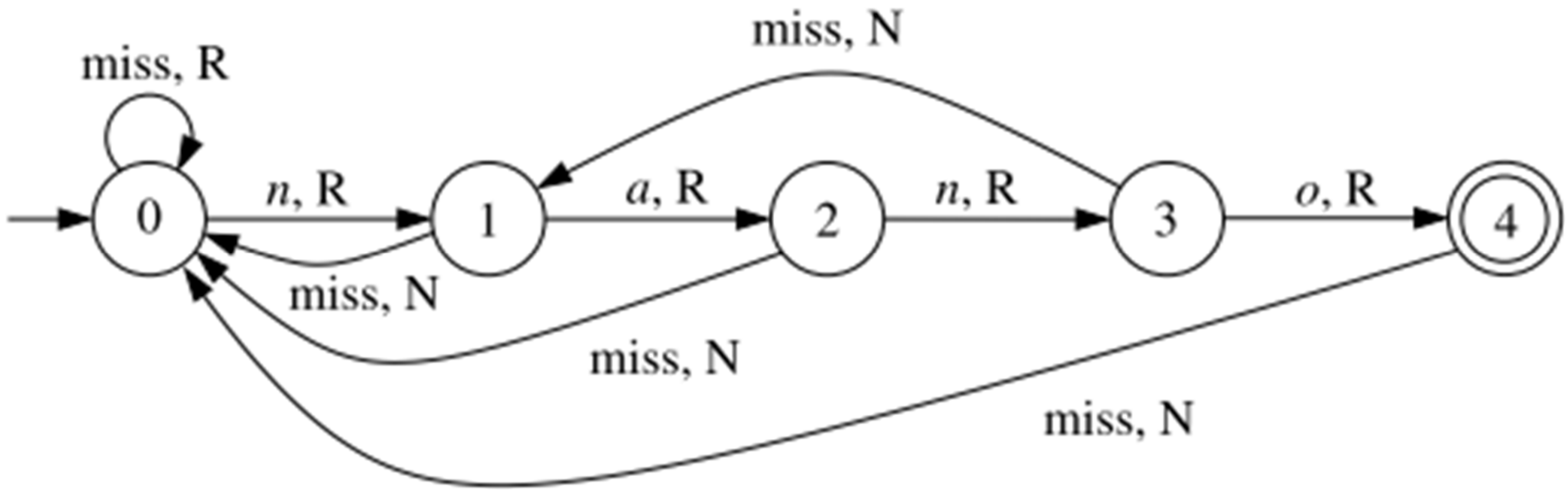
# Lazy DFAs

- We introduce a new data structure: lazy DFAs. We construct a lazy DFA for $\Sigma^* p$ with $m$ states and $2m$ transitions.
- Lazy DFAs: automata that read the input from a tape by means of a reading head that can move one cell to the right or stay put
- DFA=Lazy DFA whose head never stays put

# Lazy DFA for $\Sigma^* p$

- By the fundamental property, the DFA $B_p$ for $\Sigma^* p$ behaves from state $S_k$ as follows:
  - If $a$ is a hit, then $\delta_B(S_k, a) = S_{k+1}$ , i.e., the DFA moves to the next state in the spine.
  - If $a$ is a miss, then $\delta_B(S_k, a) = \delta_B(t(S_k), a)$, i.e., the DFA moves to the same state it would move to if it were in state $t(S_k)$.
- When $a$ is a miss for $S_k$, the lazy automaton moves to state $t(S_k)$ without advancing the head. In other words, it „delegates" doing the move to $t(S_k)$
- So the lazyDFA behaves the same for all misses.


other
n
n
n
0
n
1, 0
a
2, 0
n
3, 1, 0
o
4, 0
other
a
other
other
other


miss, R
miss, N
0
n, R
1
a, R
2
n, R
3
o, R
4
miss, N
miss, N
miss, N

- Formally,
  - $\delta_C(S_k, a) = (S_{k+1}, R)$ if $a$ is a hit
  - $\delta_C(S_k, a) = (t(S_k), N)$ if $a$ is a miss
- So the lazy DFA has $m + 1$ states and $2m$ transitions, and can be constructed in $O(m)$ space.

- Running the lazy DFA on the text takes $O(n + m)$ time:
  - For every text letter we have a sequence of „stay put“ steps followed by a „right“ step. Call it a macrostep.
  - Let $S_{j_i}$ be the state after the $i$-th macrostep. The number of steps of the $i$-th macrostep is at most $j_{i-1} - j_i + 2$ .

  So the total number of steps is at most

$$\sum_{i=1}^{n} (j_{i-1} - j_i + 2) = j_0 - j_n + 2n \leq m + 2n$$

# Computing $Miss$

- For the $O(m + n)$ bound it remains to show that the lazy DFA can be constructed in $O(m)$ time.
- Let $Miss(k)$ be the head of the state reached from $S_k$ by a miss.
- It is easy to compute each of $Miss(0), \dots, Miss(m)$ in $O(m)$ time, leading to a $O(n + m^2)$ time algorithm.
- Already good enough for almost all purposes. But, can we compute all of $Miss(0), \dots, Miss(m)$ together in time $O(m)$? Looks impossible!
- It isn‘t though ...

$$
miss(S_i) = \begin{cases} S_0 & \text{if } i = 0 \text{ or } i = 1 \\ \delta_B(miss(S_{i-1}), b_i) & \text{if } i > 1 \end{cases}
$$

$$
\delta_B(S_j, b) = \begin{cases} S_{j+1} & \text{if } b = b_{j+1} \text{ (hit)} \\ S_0 & \text{if } b \neq b_{j+1} \text{ (miss) and } j = 0 \\ \delta_B(miss(S_j), b) & \text{if } b \neq b_{j+1} \text{ (miss) and } j \neq 0 \end{cases}
$$

*Miss*($p$)

**Input:** word pattern $p = b_1 \cdots b_m$.

**Output:** heads of targets of miss transitions.

```
Miss(0) ← 0; Miss(1) ← 0
for i ← 2, . . . , m do
    Miss(i) ← DeltaB( Miss(i − 1) , b_i )
```

*DeltaB*($j, b$)

**Input:** number $j \in \{0, \ldots, m\}$, letter $b$.

**Output:** head of the state $\delta_B(S_j, b)$.

```
while b ≠ b_{j+1} and j ≠ 0 do  j ← Miss(j)
if b = b_{j+1} then return j + 1
else return 0
```

*Miss*($p$)

**Input:** word pattern $p = b_1 \cdots b_m$.

**Output:** heads of targets of miss transitions.

$Miss(0) \leftarrow 0$; $Miss(1) \leftarrow 0$

**for** $i \leftarrow 2, \ldots, m$ **do**

$\quad Miss(i) \leftarrow DeltaB(\, Miss(i-1)\,,\, b_i\,)$

*DeltaB*($j, b$)

**Input:** number $j \in \{0, \ldots, m\}$, letter $b$.

**Output:** head of the state $\delta_B(S_j, b)$.

**while** $b \neq b_{j+1}$ **and** $j \neq 0$ **do** $j \leftarrow Miss(j)$

**if** $b = b_{j+1}$ **then return** $j + 1$

**else return** $0$

- All calls to *DeltaB* lead together to $O(m)$ iterations of the while loop.
- The call $DeltaB(Miss(i-1), b\_i)$ executes at most $Miss(i-1) - (Miss(i) - 1)$ iterations.

- Total number of iterations:

$$\sum_{i=2}^{m}(Miss(i-1) - Miss(i) + 1)$$

$$\leq \; Miss(1) - Miss(m) + m$$

$$\leq m$$