

# Strängar

# Teckenkodning

## *ASCII - American Standard Code for Information Interchange*

- Tecken kodas med 7 bitar
- Exempel:

$$a = 97_{10} = 1100001_2$$

- Det enda språket som ASCII kan koda är Engelska (a-z)
- 7 bitar eftersom åttonde biten användes för feldetektering i överföringar (parity bit)
- Många kontrolltecken som inte används längre (33 stycken totalt)
- Vissa kontrolltecken har bytt betydelse
- T.ex. NUL (0) användes bl.a. för att reservera utrymme på hålkort som kunde fyllas i senare

- Flera 8-bitars teckenkodningar har utvecklats för att utöka ASCII, men de var sällan kompatibla med varandra
- T.ex. i ISO/IEC 646 finns å, ä och ö med
- 1980 påbörjades arbetet med *Unicode-standard*

# Unicode

- Unicode-standarden har som mål att kunna koda alla världens språk
- Definierar över 150000 *codepoints*, t.ex:

å = U+00E5, 指 = U+6307, 😊 = U+1F603

- Numera standard på internet

- Standarden definierar även flera olika kodningar
- UTF-8, UTF-16 och UTF-32
- Specificerar hur enskilda codepoints ska koda binärt

- UTF-8 är den vanligast kodningen av Unicode
- Det är en *längvarierande* textkodning, 1-4 *oktetter* (bytes)
- Bakåtkompatibel med ASCII (första 127 kodningarna är identiska)
- Enkel att iterera över
- Enkelt att hitta början på ett tecken

U+0000 - U+007F	0yyyzzzz			
U+0080 - U+07FF	110xxxxy	10yyzzzz		
U+0800 - U+FFFF	1110www	10xxxxy	10yyzzzz	
U+010000 - U+10FFFF	11110uvv	10vvwww	10xxxxy	10yyzzzz

- UTF-16 liknar UTF-8 men använder en eller två dubbeloktetter
- UTF-32 kodas i 32 bitar och motsvarar codepoints direkt
- Har fördelen att det går att indexera

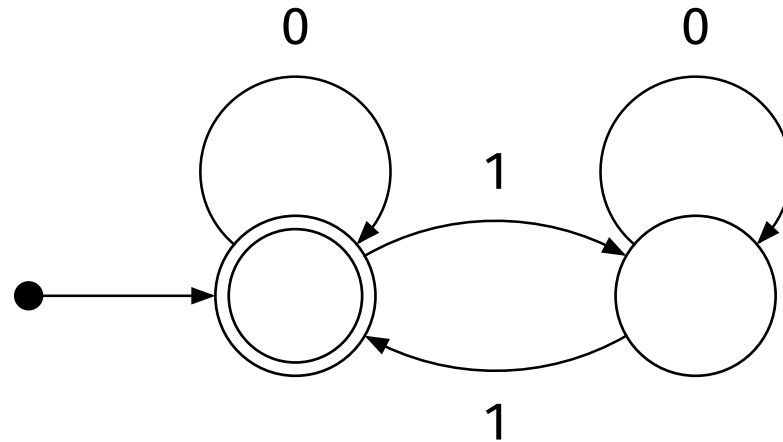
- Unicodestandarden definierar även hur man omvandlar versaler till gemener
- T.ex.  $\beta \rightarrow ss$

- Hur ska vi sortera följande lista i alfabetisk ordning?
  - Stockholm
  - Aarhus
  - Zürich

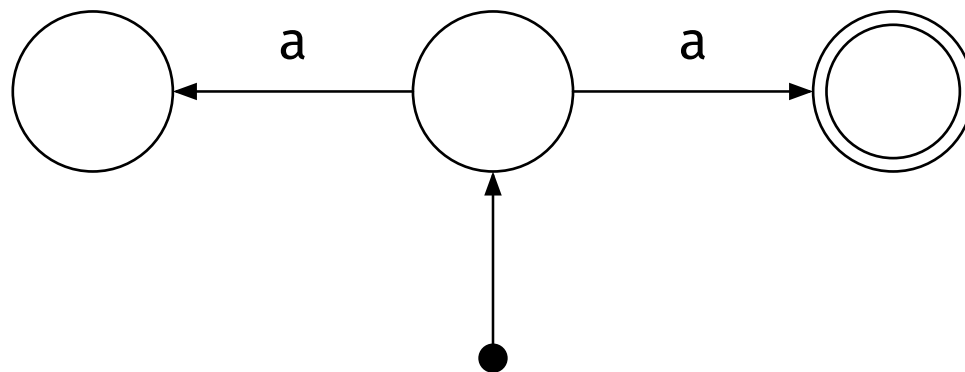
# Automater

- Automater består av olika tillstånd med övergångar
- Vissa tillstånd är *accepterande*
- Kan användas för att känna igen *reguljära språk*

- För varje övergång konsumeras en symbol
- Exempel: automat som känner igen det binära språket med ett jämnt antal ettor:



- Föregående automat var *deterministisk* (DFA)
- För varje tillstånd och varje symbol finns högst en övergång att välja
- En automat kan också vara *icke-deterministisk* (NFA). Exempel:



# Reguljära uttryck

(regex)

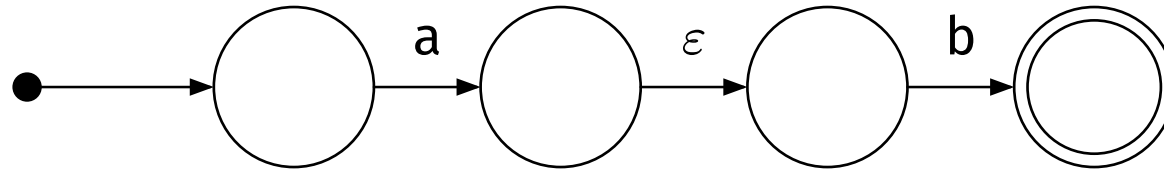
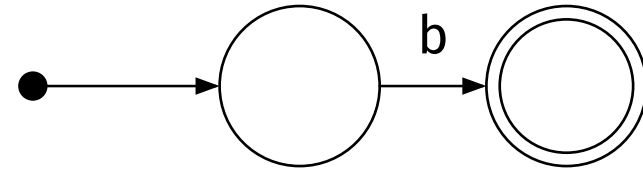
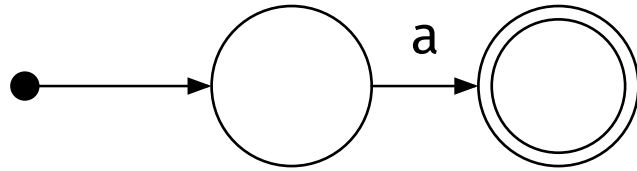
- Reguljära uttryck är ett sätt att beskriva reguljära språk
- De grundläggande operationerna är:
  - **Konkatenering:**  $abc$  matchar  $abc$
  - **Alternativ:**  $dag|natt$  matchar  $dag$ ,  $natt$
  - **Uppprepning** (Kleene closure):  $ab^*$  matchar  $a$ ,  $ab$ ,  $abb...$
  - **Gruppering:**  $a(b|c)d$  matchar  $abd$ ,  $acd$
- Dessa operationer räcker för att beskriva *alla* reguljära språk...
- ...men i praktiken har fler operationer tillkommit ( $?$ ,  $+$ ,  $[...]$ )

- När reguljära uttryck implementeras i programvara omvandlas de oftast till ändliga automater
- Det sker vanligtvis i följande steg:
  1. Omvandla uttrycket till en NFA (*Thompsons construction*)
  2. Omvandla NFA:n till en DFA (*Subset/powerset construction*)
  3. Minimera DFA:n (*Hopcrofts algorit*m)
- Det sista steget är en valfri optimering som vi inte går igenom

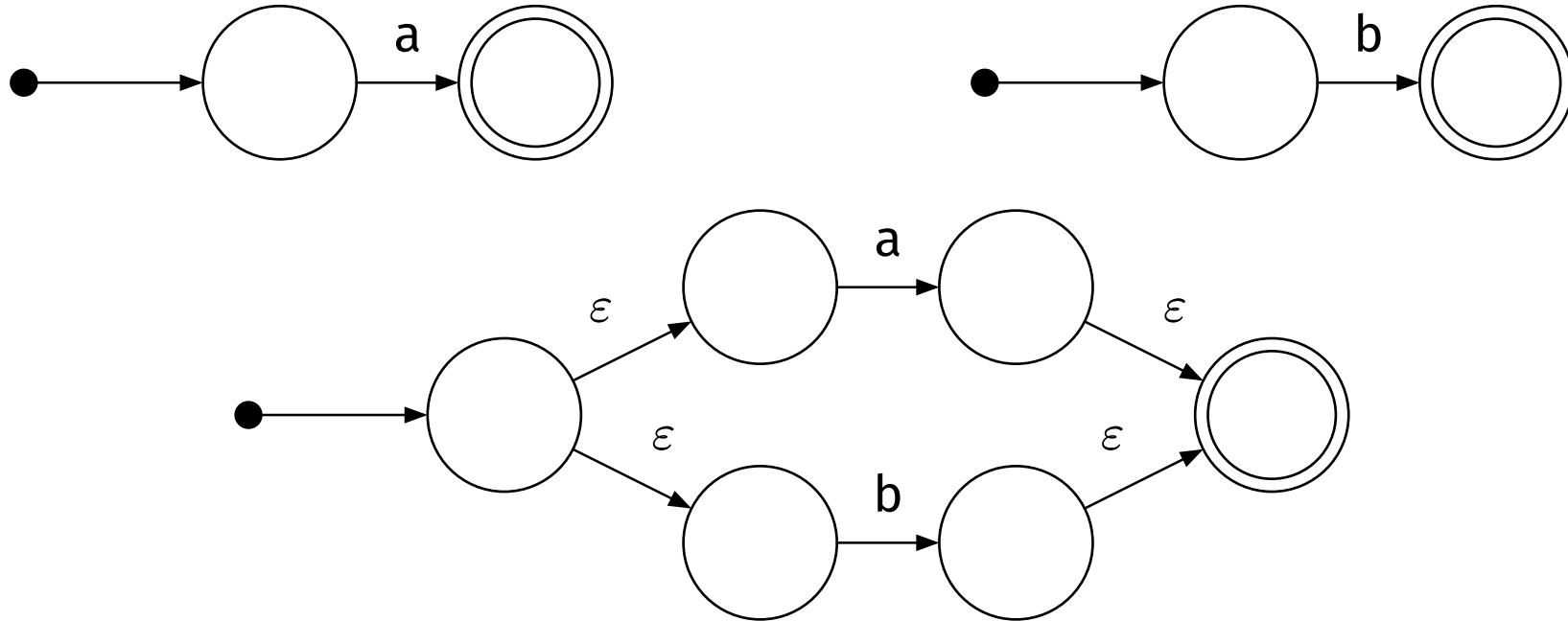
# Thompson's construction

- När vi konstruerar en NFA tillåter vi  $\varepsilon$ -övergångar.
- $\varepsilon$ -övergångar konsumerar ingen symbol

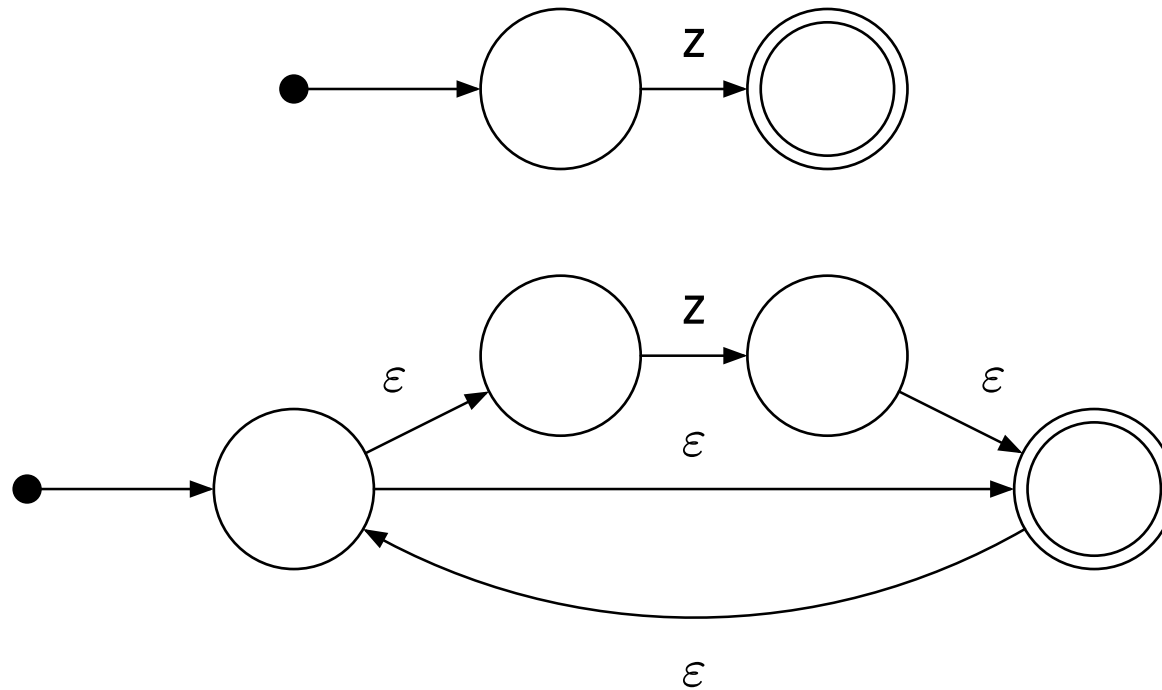
- Exempeluttryck:  $ab$



- Exempeluttryck:  $a \mid b$



- Exempeluttryck:  $Z^*$

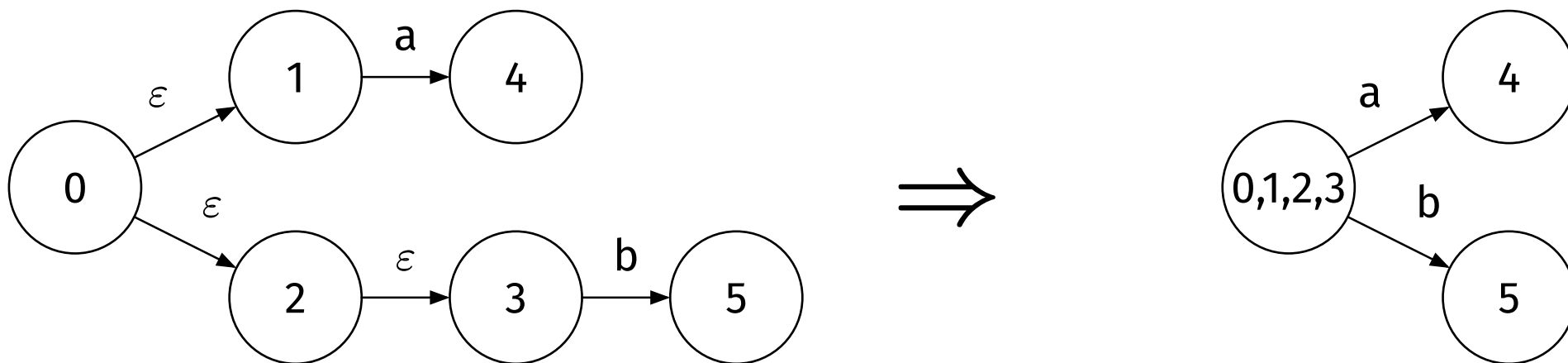


**Exempel på tavlan:**

**$(ab)^* \mid cd$**

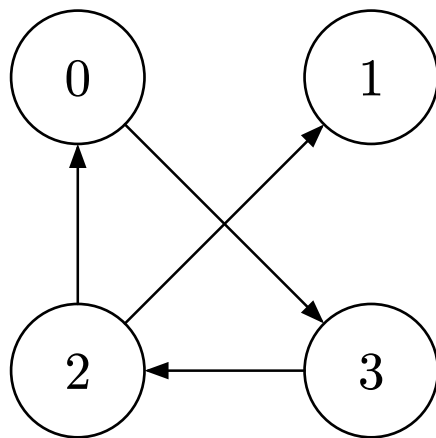


- Tillstånd i DFA:n motsvarar delmängder av tillstånden i NFA:n
- Vi slår ihop alla tillstånd som kan nås med en eller fler  $\epsilon$ -övergångar
- Vi slår även ihop tillstånd som nås med samma symbol



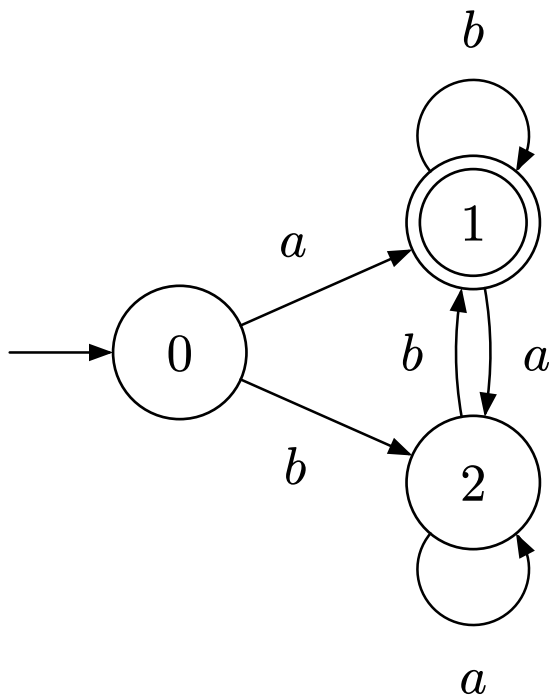
# Exekvering av NFA

Vi representerar kanter mellan tillstånd med en grannmatrix



$$\begin{array}{l} \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} \rightarrow \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \\ \uparrow & \uparrow & \uparrow & \uparrow \\ \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \end{array} \end{array} \iff \begin{array}{l} 0 \rightarrow 3 \\ 2 \rightarrow 0 \\ 2 \rightarrow 1 \\ 3 \rightarrow 2 \end{array}$$

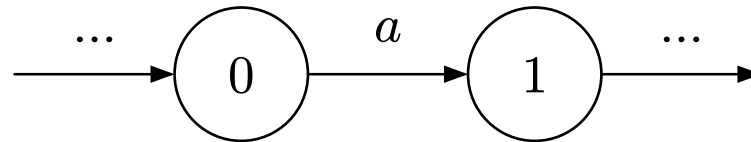
- I en automat gäller olika kanter för olika indatasymboler
- En grannmatris för varje indatasymbol



$$M(a) = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

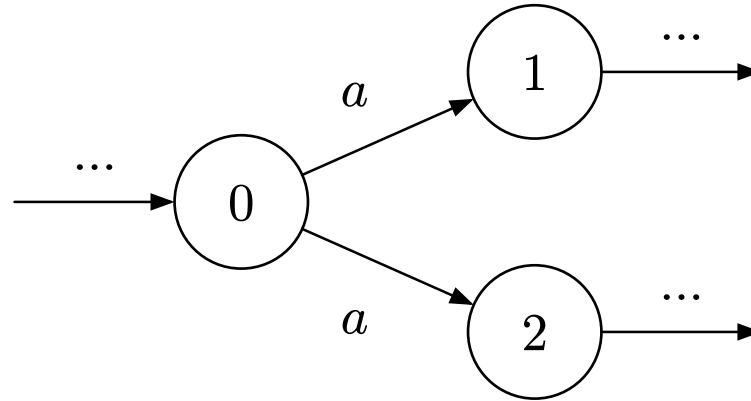
$$M(b) = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

I en DFA stegar vi genom grafen från ett tillstånd till nästa



read 'a' in state 0  
→ go to state 1

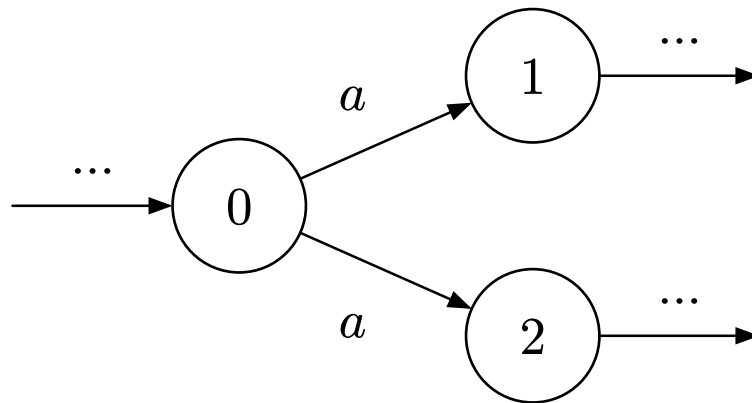
Hur gör vi med en NFA?



read 'a' in state 0

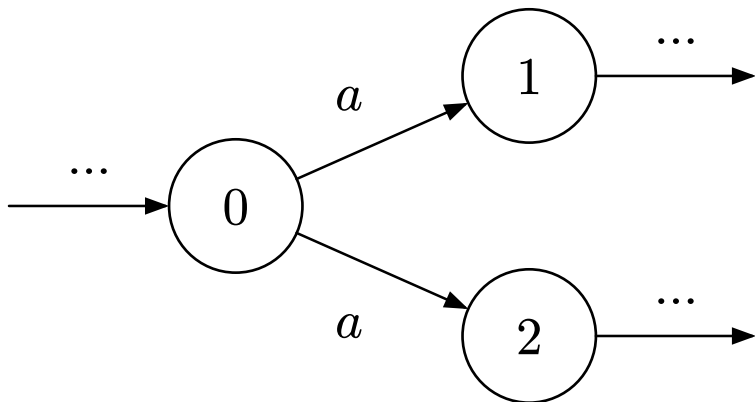
→ ?

- Istället för ett enda state har vi en "superposition"
- En vektor som räknar antalet positioner för varje tillstånd



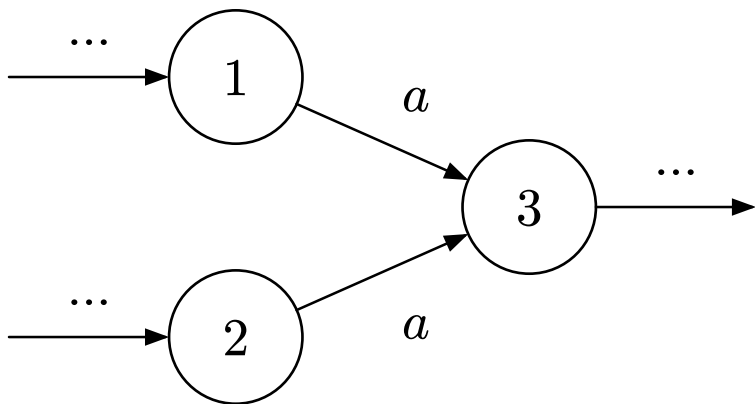
read 'a' in state  $[1, 0, 0, \dots]$   
 $\rightarrow$  go to state  $[0, 1, 1, \dots]$

- I början har vektorn bara en etta i starttillståndet:  $[1 \ 0 \ 0 \ \dots]$
- Ett steg görs med multiplikation av tillståndvektorn och grannmatrisen för nästa indatasymbol



$$\begin{aligned}\vec{s}M(a) &= [1 \ 0 \ 0 \ \dots] \begin{bmatrix} 0 & 1 & 1 & \dots \\ \vdots & \ddots & & \end{bmatrix} \\ &= [0 \ 1 \ 1 \ \dots]\end{aligned}$$

- Vi kan ha flera positioner i samma tillstånd
- Egentligen vill vi bara veta om vi är i ett tillstånd eller inte



$$\vec{s}M(a) = [0 \ 1 \ 1 \ 0 \ \dots] \begin{bmatrix} & & \dots & & \\ 0 & 0 & 0 & 1 & \dots \\ 0 & 0 & 0 & 1 & \dots \\ \vdots & & & \ddots & \end{bmatrix}$$
$$= [0 \ 0 \ 0 \ 2 \ \dots]$$

- Vi kan göra en enklare *max-min-matrismultiplikation*

Vanlig skalärprodukt med addition och multiplikation:

$$\mathbf{a} \cdot \mathbf{b} = a_1 b_1 + a_2 b_2 + \dots$$

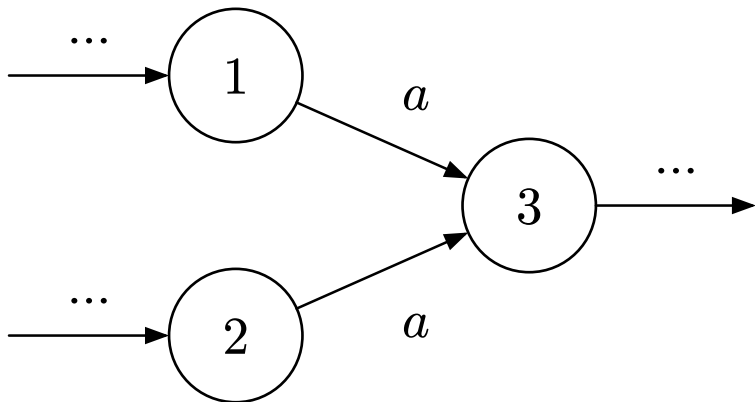
max-min-skalärprodukt:

$$\mathbf{a} \cdot \mathbf{b} = \max(\min(a_1, b_1), \min(a_2, b_2), \dots)$$

(en matrisprodukt är en matris med skalärprodukter av rader och kolumner)

- Om vi bara använder 0 och 1 i våra matriser är detta ekvivalent med *boolesk matrismultiplikation* ( $A \odot B$ )
- $\max$  är ekvivalent med OR,  $\min$  är ekvivalent med AND

$$a \cdot b = (a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee \dots$$



$$\vec{s} \odot M(a) = [0 \ 1 \ 1 \ 0 \ \dots] \begin{bmatrix} \dots & & & & \\ 0 & 0 & 0 & 1 & \dots \\ 0 & 0 & 0 & 1 & \dots \\ \vdots & & & \ddots & \end{bmatrix}$$

$$= [0 \ 0 \ 0 \ 1 \ \dots]$$

- Vi kan lagra tillståndsvektorer och matriskolumner som bitvektorer
- t.ex.  $[0 \ 1 \ 1 \ 0 \ 1] = 10110_2$
- Med högst 64 tillstånd kan en vektor eller matriskolumn vara en u64
  - För fler tillstånd kan vi stränga ihop flera ints eller använda ett bignum-bibliotek/språk
- Matrisprodukt görs med bitoperationer

```
def step(state, m):  
    new_state = 0  
    for i in range(len(m)):  
        if state & m[i] != 0:  
            new_state |= (1 << i)  
    return new_state
```

```
def run(start, accept, Ms):  
    state = start  
    for sym in sys.stdin.read():  
        state = step(state, Ms[sym])  
    return state & accept != 0
```

- Kan tolkas som deterministisk vandring genom graf av tillståndsvektorer
- Den grafen är *implicit*; vi lagrar inte alla möjliga tillståndsvektorer eller kanter mellan dem
- Om vi konstruerar en *explicit* graf har vi omvandlat vår NFA till en DFA
  - Varje NFA har därmed en mostvarande DFA; NFA och DFA är ekvivalenta
  - För en NFA med  $n$  tillstånd kan den explicita grafen av tillståndsvektorer ha upp till  $2^n$  noder

## Varför?

- Enkelt att implementera
- Snabbare att kompilera än en DFA
- Kan kräva mindre minne, mostvarande DFA har i värsta fall  $2^n$  noder
- I vissa fall även snabbare att köra
  - Optimeringar såsom state caching (typ JIT för regex) kan läggas till
- Flexibelt, lättare att lägga till utökad regex
- Behöver dock vara  $\varepsilon$ -fri precis som DFA

- Skriv ett en regex-implementation med stöd för UTF-8
- Kan vara ett kommandoradsverktyg (som grep)
- Annars (typ bibliotek): skriv *tester*!
- Ska ha stöd för konkatenering, alternativ, upprepning och gruppering
- *Valfritt*: +, ?, \, [a-z] och case insensitive
- UTF-8 ska avkodas av er
- Det går bra att samarbeta i par
- Lägg koden i ett repo med namnet <kth\_id>-regex

```
<regex> ::= <alternation> ;
```

```
<alternation> ::= <concatenation> | <concatenation> "|" <alternation> ;
```

```
<concatenation> ::= <kleene_closure> | <kleene_closure> <concatenation> ;
```

```
<kleene_closure> ::= <atom> | <atom> "*" ;
```

```
<atom> ::= CHARACTER | "(" <alternation> ")" ;
```