String Matching

Simon Moura

26 february 2019

University College London
s.moura@ucl.ac.uk
Slides: https://github.com/simmou/ucl_algo

Problem definition

The string matching problem

We consider a text T and a pattern P, where

- A **text** $T = t_1, t_2, \ldots, t_n$ is an array of n characters
- A **pattern** $P = p_1, p_2, \ldots, t_m$ is an array of m consecutive characters
- $n > m$

The goal is to find ALL occurences of pattern P in text T.

The string matching problem

We consider a text T and a pattern P, where

- A **text** $T = t_1, t_2, \ldots, t_n$ is an array of n characters
- A **pattern** $P = p_1, p_2, \ldots, t_m$ is an array of m consecutive characters
- $n > m$

The goal is to find ALL occurences of pattern P in text T.

We say that the pattern P occurs with **shift** s in text T if the **substring** of T that starts at $t_{s+1} = P$

- i.e. $t_{s+1} = p_1, t_{s+2} = p_2, \ldots, t_{s+m} = p_m$

The string matching problem

We consider a text T and a pattern P, where

- A **text** $T = t_1, t_2, \ldots, t_n$ is an array of n characters
- A **pattern** $P = p_1, p_2, \ldots, t_m$ is an array of m consecutive characters
- $n > m$

The goal is to find ALL occurences of pattern P in text T.

We say that the pattern P occurs with **shift** s in text T if the **substring** of T that starts at $t_{s+1} = P$

- i.e. $t_{s+1} = p_1, t_{s+2} = p_2, \ldots, t_{s+m} = p_m$

Concrete examples:

- The pattern $P = ell$ appears in the text $T = Hello, world$ with a shift 1.

The string matching problem

We consider a text T and a pattern P, where

- A **text** $T = t_1, t_2, \ldots, t_n$ is an array of n characters
- A **pattern** $P = p_1, p_2, \ldots, t_m$ is an array of m consecutive characters
- $n > m$

The goal is to find ALL occurences of pattern P in text T.

We say that the pattern P occurs with **shift** s in text T if the **substring** of T that starts at $t_{s+1} = P$

- i.e. $t_{s+1} = p_1, t_{s+2} = p_2, \ldots, t_{s+m} = p_m$

Concrete examples:

- The pattern $P = ell$ appears in the text $T = Hello, world$ with a shift 1.
- The pattern $P = Helo$ does not appear in the text T.

Applications: Finding a pattern in a text

Computer science page on wikipedia (first two paragraphs)

Computer science is the study of processes that interact with data and that can be represented as data in the form of programs. It enables the use of algorithms to manipulate, store, and communicate digital information. A computer scientist studies the theory of computation and the practice of designing software systems.

Its fields can be divided into theoretical and practical disciplines. Computational complexity theory is highly abstract, while computer graphics emphasizes real-world applications. Programming language theory considers approaches to the description of computational processes, while computer programming itself involves the use of programming languages and complex systems. Human–computer interaction considers the challenges in making computers useful, usable, and accessible.

How much time does the **pattern** "computer“ appears in these two paragraphs?

Applications: Finding a pattern in a text

Computer science page on wikipedia (first two paragraphs)

Computer science is the study of processes that interact with data and that can be represented as data in the form of programs. It enables the use of algorithms to manipulate, store, and communicate digital information. A computer scientist studies the theory of computation and the practice of designing software systems.

Its fields can be divided into theoretical and practical disciplines. Computational complexity theory is highly abstract, while computer graphics emphasizes real-world applications. Programming language theory considers approaches to the description of computational processes, while computer programming itself involves the use of programming languages and complex systems. Human–computer interaction considers the challenges in making computers useful, usable, and accessible.

How much time does the **pattern** ”computer“ appears in these two paragraphs?

Applications: DNA matching

Find a pattern in a DNA sequence:

- T = GTGCTATGCTGATGCTGACTTATATGCTACGTTCGGCTATC

How much time does the **pattern** GCTA appears in T?

Applications: DNA matching

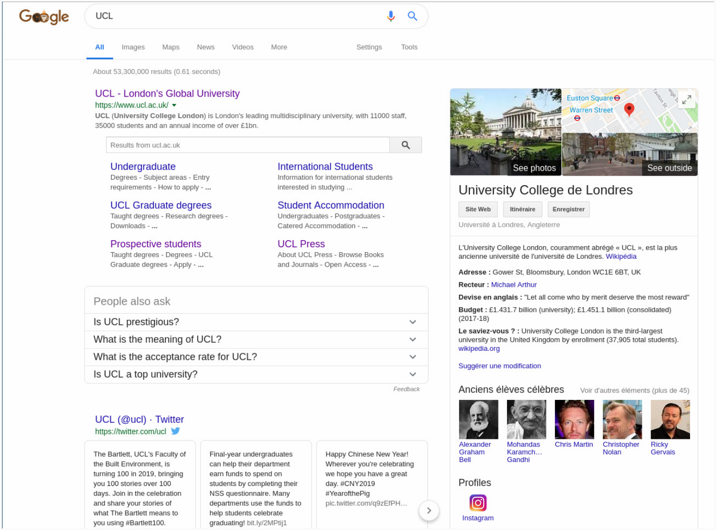
Find a pattern in a DNA sequence:

- T = GT**GCTA**TGCTGATGCTGACTTATAT**GCTA**CGTTCG**GCTA**TC

How much time does the **pattern** GCTA appears in T?

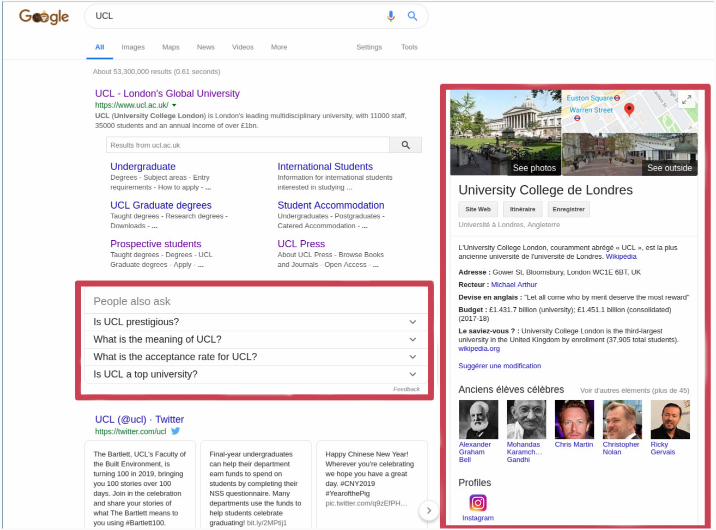
Applications: Web page relevance to queries

String matching can also be used as a first preprocessing step to provide pages relevant to queries.



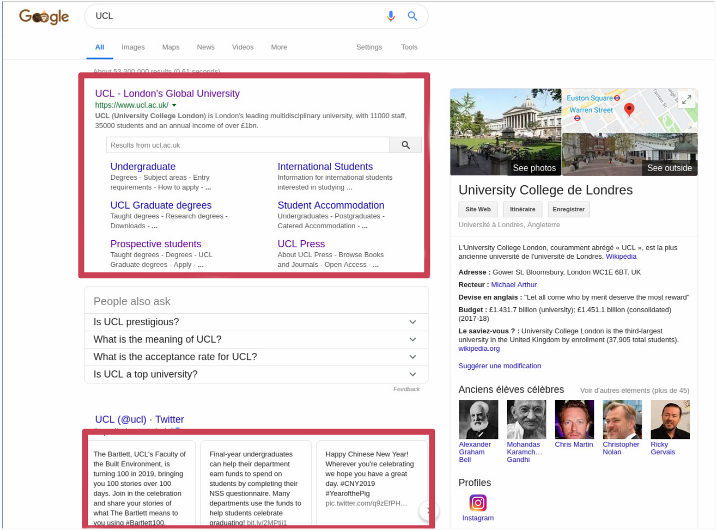
Applications: Web page relevance to queries

String matching can also be used as a first preprocessing step to provide pages relevant to queries.



Applications: Web page relevance to queries

String matching can also be used as a first preprocessing step to provide pages relevant to queries.



Naive solution to string matching

Building a naive algorithm

How much time and where does the pattern AABA appears in the text?

- Text: $T = AABAACAADAABAABA$
- Pattern: $P = AABA$

Building a naive algorithm

How much time and where does the pattern AABA appears in the text?

- Text: $T = AABAACAADAABAABA$
- Pattern: $P = AABA$

Shift	0															
T	A	A	B	A	A	C	A	A	D	A	A	B	A	A	B	A
P	A	A	B	A												

Building a naive algorithm

How much time and where does the pattern AABA appears in the text?

- Text: $T = AABAACAADAABAABA$
- Pattern: $P = AABA$

Shift		1														
T	A	A	$\cancel{B}$	A	A	C	A	A	D	A	A	B	A	A	B	A
P		A	$\cancel{A}$	B	A											

Building a naive algorithm

How much time and where does the pattern AABA appears in the text?

- Text: $T = AABAACAADAABAABA$
- Pattern: $P = AABA$

Shift			2													
T	A	A	$\cancel{B}$	A	A	C	A	A	D	A	A	B	A	A	B	A
P			$\cancel{A}$	A	B	A										

Building a naive algorithm

How much time and where does the pattern AABA appears in the text?

- Text: $T = AABAACAADAABAABA$
- Pattern: $P = AABA$

Shift				3												
T	A	A	B	A	A	$\cancel{C}$	A	A	D	A	A	B	A	A	B	A
P				A	A	$\cancel{B}$	A									

Building a naive algorithm

How much time and where does the pattern AABA appears in the text?

- Text: $T = AABAACAADAABAABA$
- Pattern: $P = AABA$

Shift	0									9			12			
T	A	A	B	A	A	C	A	A	D	A	A	B	A	A	B	A
P	A	A	B	A						A	A	B	A	A	B	A

P **occurs in** T **at shifts** $0, 9$ **and** 12**.**

Naive algorithm

```
NAIVE-STRING-MATCHING(T, P):
    n = length(T)
    m = length(P)
    for i in [0, n − m + 1] do
        for j in [1, m] do
            if T[i + j] != P[j] then
                break
            end if
        end for
        if j == m then
            print "Pattern found at shift ", i
        end if
    end for
```

Naive algorithm

```
NAIVE-STRING-MATCHING(T, P):
    n = length(T)
    m = length(P)
    for i in [0, n − m + 1] do
        for j in [1, m] do
            if T[i + j] != P[j] then
                break
            end if
        end for
        if j == m then
            print "Pattern found at shift ", i
        end if
    end for
```

Naive algorithm

```
NAIVE-STRING-MATCHING(T, P):
    n = length(T)
    m = length(P)
    for i in [0, n − m + 1] do
        for j in [1, m] do
            if T[i + j] != P[j] then
                break
            end if
        end for
        if j == m then
            print "Pattern found at shift ", i
        end if
    end for
```

Naive algorithm

```
NAIVE-STRING-MATCHING(T, P):
    n = length(T)
    m = length(P)
    for i in [0, n − m + 1] do
        for j in [1, m] do
            if T[i + j] != P[j] then
                break
            end if
        end for
        if j == m then
            print "Pattern found at shift ", i
        end if
    end for
```

Naive algorithm complexity

- **Best case scenario:** the first character of the pattern does not appear in the text:
  - $T = AABABABABABABABABA$
  - $P = ZABA$
  - Best case **complexity:** $\mathcal{O}(m)$

Naive algorithm complexity

- **Best case scenario:** the first character of the pattern does not appear in the text:
  - $T = AABABABABABABABABA$
  - $P = ZABA$
  - Best case **complexity:** $\mathcal{O}(m)$

- **Worst case scenario:** either all characters of the pattern match the full text:
  - $T = AAAAAAAAAAAAAAAAAAA$
  - $P = AAA$

Naive algorithm complexity

- **Best case scenario:** the first character of the pattern does not appear in the text:
  - $T = AABABABABABABABABA$
  - $P = ZABA$
  - Best case **complexity:** $\mathcal{O}(m)$

- **Worst case scenario:** either all characters of the pattern match the full text:
  - $T = AAAAAAAAAAAAAAAAAAA$
  - $P = AAA$

  or only the last characters of both strings are the same:
  - $T = AAAAAAAAAAAAAAAAAAZ$
  - $P = AAZ$

  Worst case **complexity:** $\mathcal{O}(m(n-m))$

Improving over the naive algorithm

What is the problem with the naive algorithm?

Improving over the naive algorithm

What is the problem with the naive algorithm?

Shift	0															
T	A	A	B	A	A	C	A	A	D	A	A	B	A	A	B	A
P	A	A	B	A												

Improving over the naive algorithm

What is the problem with the naive algorithm?

Shift		1														
T	A	A	~~B~~	A	A	C	A	A	D	A	A	B	A	A	B	A
P		A	~~A~~	B	A											

Improving over the naive algorithm

What is the problem with the naive algorithm?

Shift			2													
T	A	A	~~B~~	A	A	C	A	A	D	A	A	B	A	A	B	A
P			~~A~~	A	B	A										

Improving over the naive algorithm

What is the problem with the naive algorithm?

Shift				3												
T	A	A	B	A	A	~~C~~	A	A	D	A	A	B	A	A	B	A
P				A	A	~~B~~	A									

Improving over the naive algorithm

What is the problem with the naive algorithm?

Shift					4											
T	A	A	B	A	A	~~C~~	A	A	D	A	A	B	A	A	B	A
P					A	~~A~~	B	A								

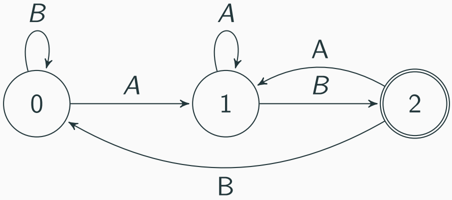
At each shift, we scan again the full substring until we find a character that does not match!

String matching with finite automaton

Finite automata definition

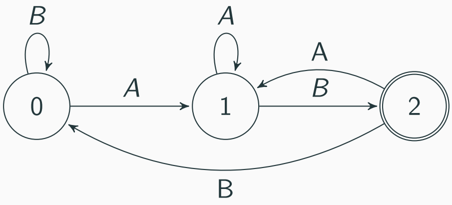
- Finite automata $M = (\mathcal{Q}, q_0, A, \Sigma, \delta)$:
  - $\mathcal{Q}$ is a finite set of **states**,
  - $q_0 \in \mathcal{Q}$ is the **start state**,
  - $A \subset \mathcal{Q}$ is the set of **accepting states**,
  - Σ is a finite **input alphabet**,
  - $\delta : (\mathcal{Q}, \Sigma) \to \mathcal{Q}$, called the **transition function of** M: determines the future state based on the current state and the current input symbol.

Example: a simple automata that matches AB



- **Starting state:** 0
- **States:** 0, 1 and 2
- **Accepting state:** 2
- **Alphabet/vocabulary**:
  $\Sigma = \{A, B\}$

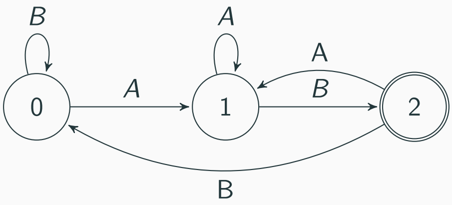
Example: a simple automata that matches AB



- **Starting state:** 0
- **States:** 0, 1 and 2
- **Accepting state:** 2
- **Alphabet/vocabulary**: $\Sigma = \{A, B\}$

- The edges of an automata allow to move from one state to another one by "consuming" a character. E.g. we move from state 0 to 1 "consuming" an A and from state 1 to 2 "consuming" a B.
- Whenever the automata arrives to an accepting state, it means we just found the pattern P in text T.

Example: a simple automata that matches AB



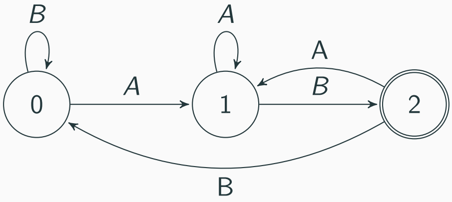
Equivalent representation of the automata:

$$\begin{array}{c c} & \begin{array}{cc} A & B \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \end{array} & \begin{pmatrix} 1 & 0 \\ 1 & 2 \\ 1 & 0 \end{pmatrix} \end{array}$$

Example on a simple text:

- $T = ABAABBB$
- $P = AB$

Example: a simple automata that matches AB



Equivalent representation of the automata:

$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \end{array}\begin{array}{c} \begin{array}{cc} A & B \end{array} \\ \begin{pmatrix} 1 & 0 \\ 1 & 2 \\ 1 & 0 \end{pmatrix} \end{array}$$

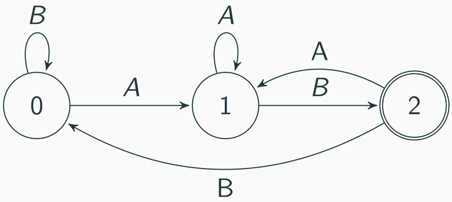
Example on a simple text:

- $T = ABAABBB$
- $P = AB$

Execution:

- **Goes from state 0 to 1 "consuming"** $t_1 = A$

Example: a simple automata that matches AB



Equivalent representation of the automata:

$$\begin{array}{c c} & \begin{array}{cc} A & B \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \end{array} & \begin{pmatrix} 1 & 0 \\ 1 & 2 \\ 1 & 0 \end{pmatrix} \end{array}$$

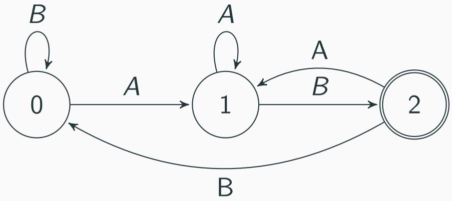
Example on a simple text:

- $T = ABAABBB$
- $P = AB$

Execution:

- **Goes from state 1 to 2 ”consuming“ $t_2 = B$, 1^{st} match at shift $s = 0$**

Example: a simple automata that matches AB



Equivalent representation of the automata:

$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \end{array} \begin{array}{c} \begin{array}{cc} A & B \end{array} \\ \begin{pmatrix} 1 & 0 \\ 1 & 2 \\ 1 & 0 \end{pmatrix} \end{array}$$

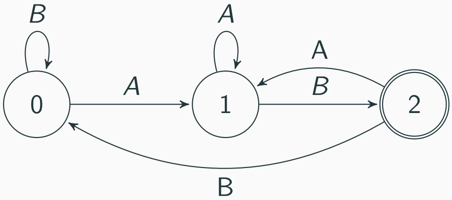
Example on a simple text:

- $T = ABAABBB$
- $P = AB$

Execution:

- **Goes from state 2 to 1 "consuming"** $t_3 = A$

Example: a simple automata that matches AB



Equivalent representation of the automata:

$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \end{array}\begin{array}{c} \begin{array}{cc} A & B \end{array} \\ \begin{pmatrix} 1 & 0 \\ 1 & 2 \\ 1 & 0 \end{pmatrix} \end{array}$$

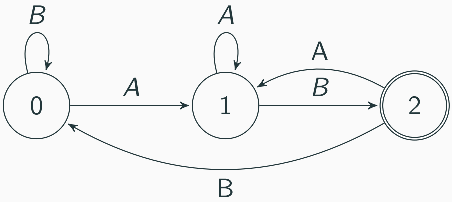
Example on a simple text:

- $T = ABAABBB$
- $P = AB$

Execution:

- **Goes from state 1 to 1 "consuming"** $t_4 = A$

Example: a simple automata that matches AB



Equivalent representation of the automata:

$$\begin{array}{c c} & \begin{array}{cc} A & B \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \end{array} & \begin{pmatrix} 1 & 0 \\ 1 & 2 \\ 1 & 0 \end{pmatrix} \end{array}$$

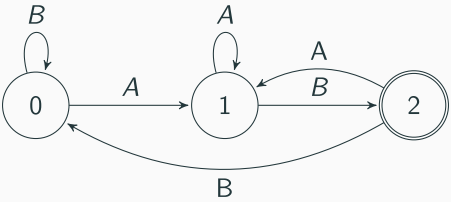
Example on a simple text:

- $T = ABAABBB$
- $P = AB$

Execution:

- **Goes from state 1 to 2 "consuming"** $t_5 = B$**, 2^{nd} match at shift** $s = 3$

Example: a simple automata that matches AB



Equivalent representation of the automata:

$$\begin{array}{c c} & \begin{array}{cc} A & B \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \end{array} & \begin{pmatrix} 1 & 0 \\ 1 & 2 \\ 1 & 0 \end{pmatrix} \end{array}$$

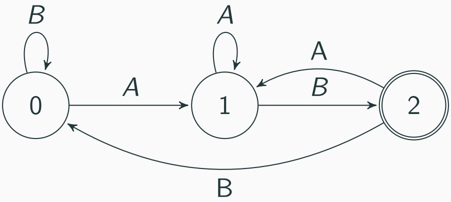
Example on a simple text:

- $T = ABAABBB$
- $P = AB$

Execution:

- **Goes from state 2 to 0 "consuming"** $t_6 = B$

Example: a simple automata that matches AB



Equivalent representation of the automata:

$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \end{array} \begin{array}{c} \begin{array}{cc} A & B \end{array} \\ \begin{pmatrix} 1 & 0 \\ 1 & 2 \\ 1 & 0 \end{pmatrix} \end{array}$$

Example on a simple text:

- $T = ABAABBB$
- $P = AB$

Execution:

- **Goes from state 0 to 0 "consuming"** $t_7 = B$

Finite automata string matcher

```
FA-STRING-MATCHER(T, next-state, m):
    n = length(T)
    state = 0
    for i in [1, n] do
        state = next-state[state, t_i]
        if state == accepting state then
            print "The pattern occurs with shift", i − m
        end if
    end for
```

Finite automata string matcher

```
FA-STRING-MATCHER(T, next-state, m):
    n = length(T)
    state = 0
    for i in [1, n] do
        state = next-state[state, t_i]
        if state == accepting state then
            print "The pattern occurs with shift", i − m
        end if
    end for
```

Finite automata string matcher

```
FA-STRING-MATCHER(T, next-state, m):
    n = length(T)
    state = 0
    for i in [1, n] do
        state = next-state[state, t_i]
        if state == accepting state then
            print "The pattern occurs with shift", i − m
        end if
    end for
```

Finite automata string matcher

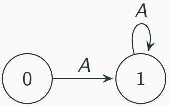
```
FA-STRING-MATCHER(T, next-state, m):
    n = length(T)
    state = 0
    for i in [1, n] do
        state = next-state[state, t_i]
        if state == accepting state then
            print "The pattern occurs with shift", i − m
        end if
    end for
```

Building a finite automata for P = ACACAGA

- $\Sigma = \{A, T, C, G\}$
- $P = ACACAGA$
- **Automata matching:**

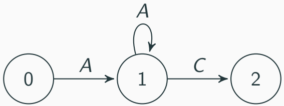
Building a finite automata for P = ACACAGA

- $\Sigma = \{A, T, C, G\}$
- $P = ACACAGA$
- **Automata matching: A**



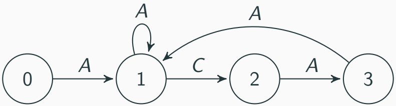
Building a finite automata for P = ACACAGA

- $\Sigma = \{A, T, C, G\}$
- $P = ACACAGA$
- **Automata matching: AC**



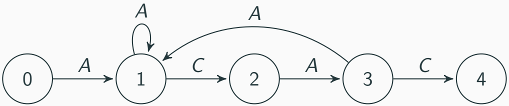
Building a finite automata for P = ACACAGA

- $\Sigma = \{A, T, C, G\}$
- $P = ACACAGA$
- **Automata matching: ACA**



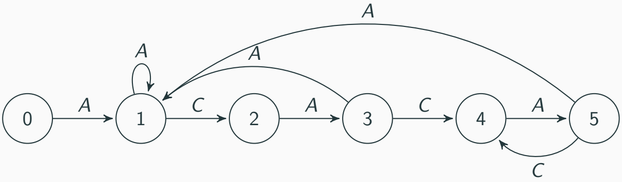
Building a finite automata for P = ACACAGA

- $\Sigma = \{A, T, C, G\}$
- $P = ACACAGA$
- **Automata matching: ACAC**



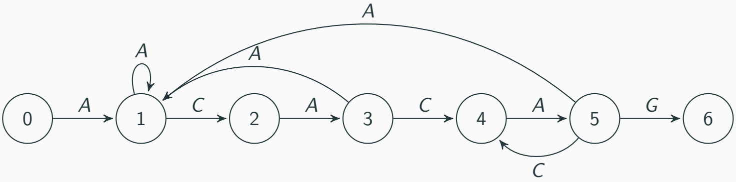
Building a finite automata for P = ACACAGA

- $\Sigma = \{A, T, C, G\}$
- $P = ACACAGA$
- **Automata matching: ACACA**



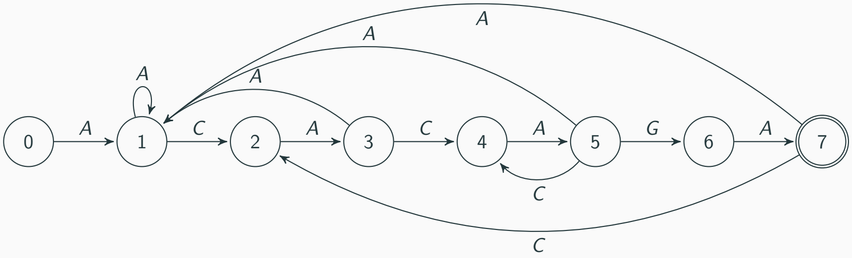
Building a finite automata for P = ACACAGA

- $\Sigma = \{A, T, C, G\}$
- $P = ACACAGA$
- **Automata matching: ACACAG**

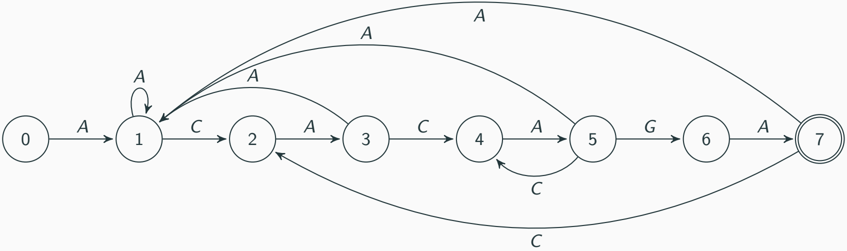


Building a finite automata for P = ACACAGA

- $\Sigma = \{A, T, C, G\}$
- $P = ACACAGA$
- **Automata matching: ACACAGA**



Building a finite automata for P = ACACAGA



$$
\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array}
\begin{array}{c}
\begin{array}{cccc} A & T & C & G \end{array} \\
\begin{pmatrix}
1 & 0 & 0 & 0 \\
1 & 0 & 2 & 0 \\
3 & 0 & 0 & 0 \\
1 & 0 & 4 & 0 \\
5 & 0 & 0 & 0 \\
1 & 0 & 4 & 6 \\
7 & 0 & 0 & 0 \\
1 & 0 & 2 & 0
\end{pmatrix}
\end{array}
$$

Building the next-state array automatically

Ideas:

1. We need to get the next state from the current state for every possible character.
2. For a character c and state s we compute the longest prefix of the pattern P that is also a suffix of $P[0, s-1]|c$.

Where:

- $P[1, s]|c$ means that we concatenate c to the string $P[1, s]$. E.g. if $P = ABC$ and $c = D$, $P[1, 3]|c = ABCD$.
- A **prefix** of a string $S = s_1, s_2, \ldots, s_n$ is a string $p = s_1, s_2, \ldots, s_k, k \leq n$.
  E.g. if $S = ABCDEF$ then $p = ABC$ is a prefix of S.
- A **suffix** of a string $S = s_1, s_2, \ldots, s_n$ is a string $p = s_k, \ldots, s_{n-1}, s_n, k \geq 1$.
  E.g. if $S = ABCDEF$ then $p = EF$ is a suffix of S.

Building the next-state array automatically

NEXT-STATE-ARRAY(P):
    m = length(P)
    for state in [0, m] **do**
        for *char* in Σ **do**
            i = min(m, state + 1)
            while $P[1, i]$ is not a suffix of pattern $P[1, state]|char$ **do**
                i = i - 1
            end while
            $\delta(\text{state}, char) = i$
        end for
    end for
    return δ

Complexity of NEXT-STATE-ARRAY: $\mathcal{O}(m^3 * |\Sigma|)$