CS 61A/CS 98-52

Mehrdad Niknami

University of California, Berkeley

Motivation

How would you find a substring inside a string?

Motivation

How would you find a substring inside a string?

Something like this? (Is this good?)

Motivation

How would you find a substring inside a string?

Something like this? (Is this good?)

```
def find(string, pattern):
    n = len(string)
    m = len(pattern)
    for i in range(n - m + 1):
        is_match = True
        for j in range(m):
            if pattern[j] != string[i + j]
                is_match = False
                break
        if is_match:
            return i
```

Motivation

How would you find a substring inside a string?

Something like this? (Is this good?)

```
def find(string, pattern):
    n = len(string)
    m = len(pattern)
    for i in range(n - m + 1):
        is_match = True
        for j in range(m):
            if pattern[j] != string[i + j]
                is_match = False
                break
        if is_match:
            return i
```

What if you were looking for a *pattern*? Like an email address?

Background

Background

Text processing has been at the heart of computer science since the **1950s**

Background

Text processing has been at the heart of computer science since the **1950s**

- Regular languages: 1950s (Kleene)

Background

Text processing has been at the heart of computer science since the **1950s**

- Regular languages: 1950s (Kleene)
- Context-free languages (CFLs): 1950s (Chomsky)

Background

Text processing has been at the heart of computer science since the **1950s**

- Regular languages: 1950s (Kleene)
- Context-free languages (CFLs): 1950s (Chomsky)
- Regular expressions (regexes) & automata: 1960s (Thompson)

Background

Text processing has been at the heart of computer science since the **1950s**

- Regular languages: 1950s (Kleene)
- Context-free languages (CFLs): 1950s (Chomsky)
- Regular expressions (regexes) & automata: 1960s (Thompson)
- LR parsing (**l**eft-to-right, **r**ightmost-derivation): 1960s (Knuth)

Background

Text processing has been at the heart of computer science since the **1950s**

- Regular languages: 1950s (Kleene)
- Context-free languages (CFLs): 1950s (Chomsky)
- Regular expressions (regexes) & automata: 1960s (Thompson)
- LR parsing (**l**eft-to-right, **r**ightmost-derivation): 1960s (Knuth)
- Context-free parsers: 1960s (Earley)

Background

Text processing has been at the heart of computer science since the **1950s**

- Regular languages: 1950s (Kleene)
- Context-free languages (CFLs): 1950s (Chomsky)
- Regular expressions (regexes) & automata: 1960s (Thompson)
- LR parsing (**l**eft-to-right, **r**ightmost-derivation): 1960s (Knuth)
- Context-free parsers: 1960s (Earley)
- String searching (Knuth-Morris-Pratt, Boyer-Moore, etc.): 1970s

Background

Text processing has been at the heart of computer science since the **1950s**

- Regular languages: 1950s (Kleene)
- Context-free languages (CFLs): 1950s (Chomsky)
- Regular expressions (regexes) & automata: 1960s (Thompson)
- LR parsing (**l**eft-to-right, **r**ightmost-derivation): 1960s (Knuth)
- Context-free parsers: 1960s (Earley)
- String searching (Knuth-Morris-Pratt, Boyer-Moore, etc.): 1970s
- Periods & critical factorizations: 1970s (Cesari-Vincent)

Background

Text processing has been at the heart of computer science since the **1950s**

- Regular languages: 1950s (Kleene)
- Context-free languages (CFLs): 1950s (Chomsky)
- Regular expressions (regexes) & automata: 1960s (Thompson)
- LR parsing (**l**eft-to-right, **r**ightmost-derivation): 1960s (Knuth)
- Context-free parsers: 1960s (Earley)
- String searching (Knuth-Morris-Pratt, Boyer-Moore, etc.): 1970s
- Periods & critical factorizations: 1970s (Cesari-Vincent)
- [...] Critical factorizations in linear complexity: **2016** (Kosolobov)

Background

Text processing has been at the heart of computer science since the **1950s**

- Regular languages: 1950s (Kleene)
- Context-free languages (CFLs): 1950s (Chomsky)
- Regular expressions (regexes) & automata: 1960s (Thompson)
- LR parsing (**l**eft-to-right, **r**ightmost-derivation): 1960s (Knuth)
- Context-free parsers: 1960s (Earley)
- String searching (Knuth-Morris-Pratt, Boyer-Moore, etc.): 1970s
- Periods & critical factorizations: 1970s (Cesari-Vincent)
- [...] Critical factorizations in linear complexity: **2016** (Kosolobov)

Research is still ongoing

Background

Text processing has been at the heart of computer science since the **1950s**

- Regular languages: 1950s (Kleene)
- Context-free languages (CFLs): 1950s (Chomsky)
- Regular expressions (regexes) & automata: 1960s (Thompson)
- LR parsing (**l**eft-to-right, **r**ightmost-derivation): 1960s (Knuth)
- Context-free parsers: 1960s (Earley)
- String searching (Knuth-Morris-Pratt, Boyer-Moore, etc.): 1970s
- Periods & critical factorizations: 1970s (Cesari-Vincent)
- [...] Critical factorizations in linear complexity: **2016** (Kosolobov)

Research is still ongoing ...apparently more in Europe?

Background

Background

Most of you will probably graduate without learning string processing.

Background

Most of you will probably graduate without learning string processing.

Instead, you'll learn how to process images and Big Data™.

Background

Most of you will probably graduate without learning string processing.

Instead, you'll learn how to process images and Big Data™.

Which makes me sad. :(You should know how to solve solved problems!

Background

Most of you will probably graduate without learning string processing.

Instead, you'll learn how to process images and Big Data™.

Which makes me sad. :(You should know how to solve solved problems!

Learn & use 100%-accurate algorithms before 85%-accurate ones!

Background

Most of you will probably graduate without learning string processing.

Instead, you'll learn how to process images and Big Data™.

Which makes me sad. :(You should know how to solve solved problems!

Learn & use 100%-accurate algorithms before 85%-accurate ones!

$O(mn)$-time `str.find(substring)` is **bad**! You can do *much* better:

Background

Most of you will probably graduate without learning string processing.

Instead, you'll learn how to process images and Big Data™.

Which makes me sad. :(You should know how to solve solved problems!

Learn & use 100%-accurate algorithms before 85%-accurate ones!

$O(mn)$-time `str.find(substring)` is **bad**! You can do *much* better:

- Good algorithms finish in $O(m+n)$ time & space (e.g. Z algorithm)

Background

Most of you will probably graduate without learning string processing.

Instead, you'll learn how to process images and Big Data™.

Which makes me sad. :(You should know how to solve solved problems!

Learn & use 100%-accurate algorithms before 85%-accurate ones!

$O(mn)$-time `str.find(substring)` is **bad**! You can do *much* better:

- Good algorithms finish in $O(m+n)$ time & space (e.g. Z algorithm)
- The best/coolest finish in $O(m+n)$ time but $O(1)$ **space**!!!

Background

Most of you will probably graduate without learning string processing.

Instead, you'll learn how to process images and Big Data™.

Which makes me sad. :(You should know how to solve solved problems!

Learn & use 100%-accurate algorithms before 85%-accurate ones!

$O(mn)$-time `str.find(substring)` is **bad**! You can do *much* better:

- Good algorithms finish in $O(m+n)$ time & space (e.g. Z algorithm)
- The best/coolest finish in $O(m+n)$ time but $O(1)$ **space**!!!

So, today, I'll teach a bit about string processing. :)

Background

Most of you will probably graduate without learning string processing.

Instead, you'll learn how to process images and Big Data™.

Which makes me sad. :(You should know how to solve solved problems!

Learn & use 100%-accurate algorithms before 85%-accurate ones!

$O(mn)$-time `str.find(substring)` is **bad**! You can do *much* better:

- Good algorithms finish in $O(m+n)$ time & space (e.g. Z algorithm)
- The best/coolest finish in $O(m+n)$ time but $O(1)$ **space**!!!

So, today, I'll teach a bit about string processing. :)

You can learn more in CS 164, CS 176, etc. (Have fun!)

Formal Languages

In formal language theory:

Formal Languages

In formal language theory:

- **Alphabet**: any set (usually a *character set*, like English or ASCII)
  $\rightarrow$ Often denoted by Σ

Formal Languages

In formal language theory:

- **Alphabet**: any set (usually a *character set*, like English or ASCII)
  $\rightarrow$ Often denoted by Σ
- **Letter**: an element in the given *alphabet*, e.g. "x"

Formal Languages

In formal language theory:

- **Alphabet**: any set (usually a *character set*, like English or ASCII)
  $\rightarrow$ Often denoted by Σ
- **Letter**: an element in the given *alphabet*, e.g. "`x`"
- **String** (or **word**): finite sequence of *letters*, e.g. "`hi`"

Formal Languages

In formal language theory:

- **Alphabet**: any set (usually a *character set*, like English or ASCII)
  → Often denoted by Σ
- **Letter**: an element in the given *alphabet*, e.g. "`x`"
- **String** (or **word**): finite sequence of *letters*, e.g. "`hi`"
- **Language**: a set of *strings*, e.g. {"`a`", "`aa`", "`aaa`", ...}
  → Often denoted by L

Formal Languages

In formal language theory:

- **Alphabet**: any set (usually a *character set*, like English or ASCII)
  → Often denoted by Σ
- **Letter**: an element in the given *alphabet*, e.g. "`x`"
- **String** (or **word**): finite sequence of *letters*, e.g. "`hi`"
- **Language**: a set of *strings*, e.g. {"`a`", "`aa`", "`aaa`", ...}
  → Often denoted by L

We might omit the quotes/braces, so we'll use the following denotations:

Formal Languages

In formal language theory:

- **Alphabet**: any set (usually a *character set*, like English or ASCII)
  $\rightarrow$ Often denoted by Σ
- **Letter**: an element in the given *alphabet*, e.g. "`x`"
- **String** (or **word**): finite sequence of *letters*, e.g. "`hi`"
- **Language**: a set of *strings*, e.g. {"`a`", "`aa`", "`aaa`", ...}
  $\rightarrow$ Often denoted by L

We might omit the quotes/braces, so we'll use the following denotations:

- ε: empty string (i.e., "")

Formal Languages

In formal language theory:

- **Alphabet**: any set (usually a *character set*, like English or ASCII)
  → Often denoted by Σ
- **Letter**: an element in the given *alphabet*, e.g. "x"
- **String** (or **word**): finite sequence of *letters*, e.g. "hi"
- **Language**: a set of *strings*, e.g. {"a", "aa", "aaa", ...}
  → Often denoted by L

We might omit the quotes/braces, so we'll use the following denotations:

- ε: empty string (i.e., "")
- $\varnothing$: empty language (i.e., empty set {})

Formal Grammars

Languages can be infinite, so we can't always list all the strings in them.

Formal Grammars

Languages can be infinite, so we can't always list all the strings in them. We therefore use **grammars** to describe languages.

Formal Grammars

Languages can be infinite, so we can't always list all the strings in them. We therefore use **grammars** to describe languages.

For example, this grammar describes $L = \{\text{""}, \text{"}\texttt{hi}\text{"}, \text{"}\texttt{hihi}\text{"}, \ldots\}$:

$$
\begin{aligned}
\texttt{S} &\rightarrow \texttt{T} \\
\texttt{T} &\rightarrow \varepsilon \\
\texttt{T} &\rightarrow \texttt{T "h" "i"}
\end{aligned}
$$

Formal Grammars

Languages can be infinite, so we can't always list all the strings in them. We therefore use **grammars** to describe languages.

For example, this grammar describes $L = \{``", ``\texttt{hi}", ``\texttt{hihi}", \ldots\}$:

$$\begin{aligned} \texttt{S} &\rightarrow \texttt{T} \\ \texttt{T} &\rightarrow \varepsilon \\ \texttt{T} &\rightarrow \texttt{T "h" "i"} \end{aligned}$$

We call `S` a **nonterminal** symbol and "`h`" a **terminal** symbol (i.e., letter).

Formal Grammars

Languages can be infinite, so we can't always list all the strings in them. We therefore use **grammars** to describe languages.

For example, this grammar describes $L = \{$"", "`hi`", "`hihi`", $\ldots\}$:

$$\begin{aligned} \texttt{S} &\rightarrow \texttt{T} \\ \texttt{T} &\rightarrow \varepsilon \\ \texttt{T} &\rightarrow \texttt{T "h" "i"} \end{aligned}$$

We call `S` a **nonterminal** symbol and "`h`" a **terminal** symbol (i.e., letter). Each line is a **production rule**, producing a **sentential form** on the right.

Formal Grammars

Languages can be infinite, so we can't always list all the strings in them. We therefore use **grammars** to describe languages.

For example, this grammar describes $L = \{\text{""}, \text{"}\texttt{hi}\text{"}, \text{"}\texttt{hihi}\text{"}, \ldots\}$:

$$\begin{aligned} \texttt{S} &\rightarrow \texttt{T} \\ \texttt{T} &\rightarrow \varepsilon \\ \texttt{T} &\rightarrow \texttt{T "h" "i"} \end{aligned}$$

We call `S` a **nonterminal** symbol and "`h`" a **terminal** symbol (i.e., letter). Each line is a **production rule**, producing a **sentential form** on the right.

To make life easier, we'll denote these by uppercase and lowercase respectively, omitting quotes and spaces when convenient.

Formal Grammars

Languages can be infinite, so we can't always list all the strings in them. We therefore use **grammars** to describe languages.

For example, this grammar describes $L = \{$"", "`hi`", "`hihi`", $\ldots\}$:

$$\texttt{S} \rightarrow \texttt{T}$$
$$\texttt{T} \rightarrow \varepsilon$$
$$\texttt{T} \rightarrow \texttt{T "h" "i"}$$

We call `S` a **nonterminal** symbol and "`h`" a **terminal** symbol (i.e., letter). Each line is a **production rule**, producing a **sentential form** on the right.

To make life easier, we'll denote these by uppercase and lowercase respectively, omitting quotes and spaces when convenient.
We then merge and simplify rules via the pipe (OR) symbol:

$$\texttt{S} \rightarrow \texttt{S hi} \mid \varepsilon$$

Regular Languages

Regular Languages

The following are **regular languages** over the alphabet Σ:

Regular Languages

The following are **regular languages** over the alphabet Σ:

- $\varnothing$

Regular Languages

The following are **regular languages** over the alphabet Σ:

- $\varnothing$
- $\{\varepsilon\}$

Regular Languages

The following are **regular languages** over the alphabet Σ:

- $\varnothing$
- $\{\varepsilon\}$
- $\{\sigma\} \quad \forall\ \sigma \in \Sigma$

Regular Languages

The following are **regular languages** over the alphabet Σ:

- $\varnothing$
- $\{\varepsilon\}$
- $\{\sigma\} \quad \forall\, \sigma \in \Sigma$
- The union $A \cup B$ of any regular languages A and B over Σ

Regular Languages

The following are **regular languages** over the alphabet Σ:

- $\varnothing$
- $\{\varepsilon\}$
- $\{\sigma\} \quad \forall\, \sigma \in \Sigma$
- The union $A \cup B$ of any regular languages A and B over Σ
- The concatenation AB of any regular languages A and B over Σ

Regular Languages

The following are **regular languages** over the alphabet Σ:

- $\varnothing$
- $\{\varepsilon\}$
- $\{\sigma\} \quad \forall\ \sigma \in \Sigma$
- The union $A \cup B$ of any regular languages A and B over Σ
- The concatenation AB of any regular languages A and B over Σ
- The repetition (Kleene star) A^* of any regular language A over Σ
  $A^* = \{\varepsilon\} \cup A \cup AA \cup AAA \cup \ldots$

Regular Languages

The following are **regular languages** over the alphabet Σ:

- $\varnothing$
- $\{\varepsilon\}$
- $\{\sigma\} \quad \forall\, \sigma \in \Sigma$
- The union $A \cup B$ of any regular languages A and B over Σ
- The concatenation AB of any regular languages A and B over Σ
- The repetition (Kleene star) A^* of any regular language A over Σ
  $A^* = \{\varepsilon\} \cup A \cup AA \cup AAA \cup \ldots$

Notice that all finite languages are regular, but not all infinite languages.

Regular Languages

The following are **regular languages** over the alphabet Σ:

- $\varnothing$
- $\{\varepsilon\}$
- $\{\sigma\} \quad \forall\, \sigma \in \Sigma$
- The union $A \cup B$ of any regular languages A and B over Σ
- The concatenation AB of any regular languages A and B over Σ
- The repetition (Kleene star) A^* of any regular language A over Σ
  $A^* = \{\varepsilon\} \cup A \cup AA \cup AAA \cup \ldots$

Notice that all finite languages are regular, but not all infinite languages.

Regular languages do not allow arbitrary "nesting" (e.g. parens).

Regular Grammars

Regular Grammars

A **regular grammar** is a grammar in which all productions have at most one nonterminal symbol, all of which appear on either the left or the right.

Regular Grammars

A **regular grammar** is a grammar in which all productions have at most one nonterminal symbol, all of which appear on either the left or the right.

In other words, this is a regular grammar:

$$\begin{aligned} \texttt{S} &\rightarrow \texttt{A b c} \\ \texttt{A} &\rightarrow \texttt{S a} \mid \varepsilon \end{aligned}$$

Regular Grammars

A **regular grammar** is a grammar in which all productions have at most one nonterminal symbol, all of which appear on either the left or the right.

In other words, this is a regular grammar:

$$\begin{aligned} \texttt{S} &\rightarrow \texttt{A b c} \\ \texttt{A} &\rightarrow \texttt{S a} \mid \varepsilon \end{aligned}$$

This is not a regular grammar (but it is *linear* and *context-free*):

$$\begin{aligned} \texttt{S} &\rightarrow \texttt{A b c} \\ \texttt{A} &\rightarrow \texttt{a S} \mid \varepsilon \end{aligned}$$

Regular Grammars

A **regular grammar** is a grammar in which all productions have at most one nonterminal symbol, all of which appear on either the left or the right.

In other words, this is a regular grammar:

$$\begin{aligned} \texttt{S} &\rightarrow \texttt{A b c} \\ \texttt{A} &\rightarrow \texttt{S a} \mid \varepsilon \end{aligned}$$

This is not a regular grammar (but it is *linear* and *context-free*):

$$\begin{aligned} \texttt{S} &\rightarrow \texttt{A b c} \\ \texttt{A} &\rightarrow \texttt{a S} \mid \varepsilon \end{aligned}$$

and neither is this (it is *context-sensitive*):

$$\begin{aligned} \texttt{S} &\rightarrow \texttt{S s} \mid \varepsilon \\ \texttt{S s} &\rightarrow \texttt{S t} \end{aligned}$$

Regular Grammars

A **regular grammar** is a grammar in which all productions have at most one nonterminal symbol, all of which appear on either the left or the right.

In other words, this is a regular grammar:

$$\begin{aligned} \texttt{S} &\rightarrow \texttt{A b c} \\ \texttt{A} &\rightarrow \texttt{S a} \mid \varepsilon \end{aligned}$$

This is not a regular grammar (but it is *linear* and *context-free*):

$$\begin{aligned} \texttt{S} &\rightarrow \texttt{A b c} \\ \texttt{A} &\rightarrow \texttt{a S} \mid \varepsilon \end{aligned}$$

and neither is this (it is *context-sensitive*):

$$\begin{aligned} \texttt{S} &\rightarrow \texttt{S s} \mid \varepsilon \\ \texttt{S s} &\rightarrow \texttt{S t} \end{aligned}$$

A language is regular iff it can be described by a regular grammar.

Regular Expressions

Regular Expressions

A **regular expression** is an easier way to describe a regular language.

Regular Expressions

A **regular expression** is an easier way to describe a regular language.
It's essentially a pattern for describing a regular language.

Regular Expressions

A **regular expression** is an easier way to describe a regular language.
It's essentially a pattern for describing a regular language.
For example, in `[abcw-z]*(1+2|3)?4\?`, we have:

Regular Expressions

A **regular expression** is an easier way to describe a regular language.
It's essentially a pattern for describing a regular language.
For example, in `[abcw-z]*(1+2|3)?4\?`, we have:

- `[abcw-z]` (a *character set*) means "either `a`, `b`, `c`, `w`, `x`, `y`, or `z`".

Regular Expressions

A **regular expression** is an easier way to describe a regular language.
It's essentially a pattern for describing a regular language.
For example, in `[abcw-z]*(1+2|3)?4\?`, we have:

- `[abcw-z]` (a *character set*) means "either `a`, `b`, `c`, `w`, `x`, `y`, or `z`".
- Asterisk (a.k.a. "Kleene star", a *quantifier*) means "zero or more"

Regular Expressions

A **regular expression** is an easier way to describe a regular language.
It's essentially a pattern for describing a regular language.
For example, in `[abcw-z]*(1+2|3)?4\?`, we have:

- `[abcw-z]` (a *character set*) means "either `a`, `b`, `c`, `w`, `x`, `y`, or `z`".
- Asterisk (a.k.a. "Kleene star", a *quantifier*) means "zero or more"
- Plus (another quantifier) means "one or more"

Regular Expressions

A **regular expression** is an easier way to describe a regular language.
It's essentially a pattern for describing a regular language.
For example, in `[abcw-z]*(1+2|3)?4\?`, we have:

- `[abcw-z]` (a *character set*) means "either `a`, `b`, `c`, `w`, `x`, `y`, or `z`".
- Asterisk (a.k.a. "Kleene star", a *quantifier*) means "zero or more"
- Plus (another quantifier) means "one or more"
- Question mark (another quantifier) means "at most one"

Regular Expressions

A **regular expression** is an easier way to describe a regular language.
It's essentially a pattern for describing a regular language.
For example, in `[abcw-z]*(1+2|3)?4\?`, we have:

- `[abcw-z]` (a *character set*) means "either `a`, `b`, `c`, `w`, `x`, `y`, or `z`".
- Asterisk (a.k.a. "Kleene star", a *quantifier*) means "zero or more"
- Plus (another quantifier) means "one or more"
- Question mark (another quantifier) means "at most one"
- Backslash ("escape") before a special character means *that character*

Regular Expressions

A **regular expression** is an easier way to describe a regular language.
It's essentially a pattern for describing a regular language.
For example, in `[abcw-z]*(1+2|3)?4\?`, we have:

- `[abcw-z]` (a *character set*) means "either `a`, `b`, `c`, `w`, `x`, `y`, or `z`".
- Asterisk (a.k.a. "Kleene star", a *quantifier*) means "zero or more"
- Plus (another quantifier) means "one or more"
- Question mark (another quantifier) means "at most one"
- Backslash ("escape") before a special character means *that character*
- Pipe (the OR symbol |) means "either", and parentheses group

Regular Expressions

A **regular expression** is an easier way to describe a regular language.
It's essentially a pattern for describing a regular language.
For example, in `[abcw-z]*(1+2|3)?4\?`, we have:

- `[abcw-z]` (a *character set*) means "either `a`, `b`, `c`, `w`, `x`, `y`, or `z`".
- Asterisk (a.k.a. "Kleene star", a *quantifier*) means "zero or more"
- Plus (another quantifier) means "one or more"
- Question mark (another quantifier) means "at most one"
- Backslash ("escape") before a special character means *that character*
- Pipe (the OR symbol |) means "either", and parentheses group

So this matches zero or more of a, b, c, w, x, y, z, followed by either nothing or by 3 or by 1's followed by 2, followed by 4 and a question mark.

Regular Expressions

[1] If you've seen backreferences: those are **not** technically valid in regexes.

Regular Expressions

Regular expressions (**regexes**) are equivalent to regular grammars[1], e.g.

$$\underbrace{\overbrace{\underbrace{\texttt{[abcw-z]}^{*}}_{\texttt{X}}\,\texttt{(1+}}^{\texttt{Y}}\texttt{2|3)?4}}_{\texttt{Z}}\backslash\texttt{?}$$

[1] If you've seen backreferences: those are **not** technically valid in regexes.

Regular Expressions

Regular expressions (**regexes**) are equivalent to regular grammars[1], e.g.

$$\underbrace{\overbrace{\underbrace{\texttt{[abcw-z]}^{*}}_{\texttt{X}}\,\texttt{(1+}}^{\texttt{Y}}\texttt{2|3)?}}_{\texttt{Z}}\,\texttt{4}\backslash\texttt{?}$$

is equivalent to

$$\begin{array}{lcl}
\texttt{S} & \rightarrow & \texttt{Z 4 ?} \\
\texttt{Z} & \rightarrow & \texttt{Y 2} \mid \texttt{X 3} \mid \varepsilon \\
\texttt{Y} & \rightarrow & \texttt{Y 1} \mid \texttt{X 1} \\
\texttt{X} & \rightarrow & \texttt{X a} \mid \texttt{X b} \mid \texttt{X c} \mid \texttt{X w} \mid \texttt{X x} \mid \texttt{X y} \mid \texttt{X z} \mid \varepsilon
\end{array}$$

[1] If you've seen backreferences: those are **not** technically valid in regexes.

Regular Expressions

Regular expressions (**regexes**) are equivalent to regular grammars[1], e.g.

$$\underbrace{\overbrace{\underbrace{\texttt{[abcw-z]}^{*}}_{\color{green}{\texttt{X}}}\,\texttt{(1+}}^{\color{blue}{\texttt{Y}}}\texttt{2|3)?}}_{\color{red}{\texttt{Z}}}\texttt{4}\backslash\texttt{?}$$

is equivalent to

$$\begin{aligned}
\texttt{S} &\rightarrow \texttt{Z 4 ?} \\
\texttt{Z} &\rightarrow \texttt{Y 2} \mid \texttt{X 3} \mid \varepsilon \\
\texttt{Y} &\rightarrow \texttt{Y 1} \mid \texttt{X 1} \\
\texttt{X} &\rightarrow \texttt{X a} \mid \texttt{X b} \mid \texttt{X c} \mid \texttt{X w} \mid \texttt{X x} \mid \texttt{X y} \mid \texttt{X z} \mid \varepsilon
\end{aligned}$$

Here, the regex is more compact. Sometimes, the grammar is smaller.

[1] If you've seen backreferences: those are **not** technically valid in regexes.

Regular Expressions

Python has a regex engine to find text matching a regex:

Regular Expressions

Python has a regex engine to find text matching a regex:

```
>>> import re
>>> m = re.match('.* ([a-z0-9._-]+)@([a-z0-9._-]+)',
                 'hello cs61a@berkeley.edu cs98-52')
>>> m
<re.Match object; span=(0, 24),
                  match='hello cs61a@berkeley.edu'>
>>> m.groups()
('cs61a', 'berkeley.edu')
```

Regular Expressions

Python has a regex engine to find text matching a regex:

```
>>> import re
>>> m = re.match('.* ([a-z0-9._-]+)@([a-z0-9._-]+)',
                 'hello cs61a@berkeley.edu cs98-52')
>>> m
<re.Match object; span=(0, 24),
                  match='hello cs61a@berkeley.edu'>
>>> m.groups()
('cs61a', 'berkeley.edu')
```

Notice that these could all be handled by `re.match`:

Regular Expressions

Python has a regex engine to find text matching a regex:

```
>>> import re
>>> m = re.match('.* ([a-z0-9._-]+)@([a-z0-9._-]+)',
                 'hello cs61a@berkeley.edu cs98-52')
>>> m
<re.Match object; span=(0, 24),
                  match='hello cs61a@berkeley.edu'>
>>> m.groups()
('cs61a', 'berkeley.edu')
```

Notice that these could all be handled by `re.match`:

- Substring search (`str.find`)

Regular Expressions

Python has a regex engine to find text matching a regex:

```
>>> import re
>>> m = re.match('.* ([a-z0-9._-]+)@([a-z0-9._-]+)',
                 'hello cs61a@berkeley.edu cs98-52')
>>> m
<re.Match object; span=(0, 24),
                  match='hello cs61a@berkeley.edu'>
>>> m.groups()
('cs61a', 'berkeley.edu')
```

Notice that these could all be handled by `re.match`:

- Substring search (`str.find`)
- Subsequence search (`re.match(".*b.*b", "abbc")`)

Regular Expressions

Python has a regex engine to find text matching a regex:

```
>>> import re
>>> m = re.match('.* ([a-z0-9._-]+)@([a-z0-9._-]+)',
                 'hello cs61a@berkeley.edu cs98-52')
>>> m
<re.Match object; span=(0, 24),
                  match='hello cs61a@berkeley.edu'>
>>> m.groups()
('cs61a', 'berkeley.edu')
```

Notice that these could all be handled by `re.match`:

- Substring search (`str.find`)
- Subsequence search (`re.match(".*b.*b", "abbc")`)

The grep tool (from ed's `g/re/p` = *global/regex/print*) does this for files.

Regular Expressions

Regular Expressions

Million-dollar question:

Million-dollar question:

How do you find text matching a regex?

Million-dollar question:

How do you find text matching a regex?

Two steps:

Regular Expressions

Million-dollar question:

How do you find text matching a regex?

Two steps:

1. Parse the regex (pattern) to "understand" its structure

Regular Expressions

Million-dollar question:

How do you find text matching a regex?

Two steps:

1. Parse the regex (pattern) to "understand" its structure
2. Use the regex to parse the actual text (corpus)

Regular Expressions

Million-dollar question:

How do you find text matching a regex?

Two steps:

1. Parse the regex (pattern) to "understand" its structure
2. Use the regex to parse the actual text (corpus)

It turns out that:

Million-dollar question:

How do you find text matching a regex?

Two steps:

1. Parse the regex (pattern) to "understand" its structure
2. Use the regex to parse the actual text (corpus)

It turns out that:

1. Step 1 is theoretically harder, but practically easier.
   (This can be done similarly to how you parsed Scheme.)

Regular Expressions

Million-dollar question:

How do you find text matching a regex?

Two steps:

1. Parse the regex (pattern) to “understand” its structure
2. Use the regex to parse the actual text (corpus)

It turns out that:

1. Step 1 is theoretically harder, but practically easier.
   (This can be done similarly to how you parsed Scheme.)
2. Step 2 is theoretically easier, but practically harder.

Million-dollar question:

How do you find text matching a regex?

Two steps:

1. Parse the regex (pattern) to "understand" its structure
2. Use the regex to parse the actual text (corpus)

It turns out that:

1. Step 1 is theoretically harder, but practically easier.
   (This can be done similarly to how you parsed Scheme.)
2. Step 2 is theoretically easier, but practically harder.

This is because we need parsing the *corpus* to be ***fast***.

Regular Expressions

Regular Expressions

How do you solve each step?

Regular Expressions

How do you solve each step?

Both steps are often done using "recursive-descent"—similarly to how your Scheme parser parsed its input.

Regular Expressions

How do you solve each step?

Both steps are often done using "recursive-descent"—similarly to how your Scheme parser parsed its input.

Basically: try every possibility recursively.
"Backtrack" on failure to try something else.

How do you solve each step?

Both steps are often done using "recursive-descent"—similarly to how your Scheme parser parsed its input.

Basically: try every possibility recursively.
"Backtrack" on failure to try something else.

Problem: Recursive-descent can take **exponential time**!

Regular Expressions

How do you solve each step?

Both steps are often done using "recursive-descent"—similarly to how your Scheme parser parsed its input.

Basically: try every possibility recursively.
"Backtrack" on failure to try something else.

Problem: Recursive-descent can take **exponential time**!
Example (where "a{3}" is shorthand for "aaa"):

How do you solve each step?

Both steps are often done using "recursive-descent"—similarly to how your Scheme parser parsed its input.

Basically: try every possibility recursively.
"Backtrack" on failure to try something else.

Problem: Recursive-descent can take **exponential time**!
Example (where "a{3}" is shorthand for "aaa"):

```
>>> re.match("(a?){25}a{25}", "a" * 25)
```

Regular Expressions

How do you solve each step?

Both steps are often done using "recursive-descent"—similarly to how your Scheme parser parsed its input.

Basically: try every possibility recursively.
"Backtrack" on failure to try something else.

Problem: Recursive-descent can take **exponential time**!
Example (where "a{3}" is shorthand for "aaa"):

```
>>> re.match("(a?){25}a{25}", "a" * 25)
```

Can we hope to parse corpora in time **linear** to their lengths?

Regular Expressions

How do you solve each step?

Both steps are often done using “recursive-descent”—similarly to how your Scheme parser parsed its input.

Basically: try every possibility recursively.
“Backtrack” on failure to try something else.

Problem: Recursive-descent can take **exponential time**!
Example (where “a{3}” is shorthand for “aaa”):

```
>>> re.match("(a?){25}a{25}", "a" * 25)
```

Can we hope to parse corpora in time **linear** to their lengths?

Yes, using finite automata.

Finite Automata

A finite automaton (FA) consists of the following (example below)[2]:

[2] Note that an FA is *not* quite the same thing as a finite-state machine (FSM).

Finite Automata

A finite automaton (FA) consists of the following (example below)[2]:

- An *input alphabet* Σ ($\{0, 1\}$ here)

[2]Note that an FA is *not* quite the same thing as a finite-state machine (FSM).

Finite Automata

A finite automaton (FA) consists of the following (example below)[2]:

- An *input alphabet* Σ ($\{0, 1\}$ here)
- A finite set of *states* S ($\{s_0, s_1, s_2\}$ here)

[2] Note that an FA is *not* quite the same thing as a finite-state machine (FSM).

Finite Automata

A finite automaton (FA) consists of the following (example below)[2]:

- An *input alphabet* Σ ($\{0, 1\}$ here)
- A finite set of *states* S ($\{s_0, s_1, s_2\}$ here)
- An *initial state* $s_0 \in S$ (s_0 here)

[2]Note that an FA is *not* quite the same thing as a finite-state machine (FSM).

Finite Automata

A finite automaton (FA) consists of the following (example below)[2]:

- An *input alphabet* Σ ($\{0, 1\}$ here)
- A finite set of *states* S ($\{s_0, s_1, s_2\}$ here)
- An *initial state* $s_0 \in S$ (s_0 here)
- A set of *accepting* (or *final*) *states* $F \subset S$ ($\{s_2\}$ here)

[2]Note that an FA is *not* quite the same thing as a finite-state machine (FSM).

Finite Automata

A finite automaton (FA) consists of the following (example below)[2]:

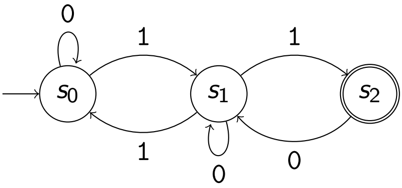
- An *input alphabet* Σ ($\{0, 1\}$ here)
- A finite set of *states* S ($\{s_0, s_1, s_2\}$ here)
- An *initial state* $s_0 \in S$ (s_0 here)
- A set of *accepting* (or *final*) *states* $F \subset S$ ($\{s_2\}$ here)
- A *transition function* $\delta : S \times \Sigma \to 2^S$ (the arrows here)

[2] Note that an FA is *not* quite the same thing as a finite-state machine (FSM).

Finite Automata

A finite automaton (FA) consists of the following (example below)[2]:

- An *input alphabet* Σ ($\{0, 1\}$ here)
- A finite set of *states* S ($\{s_0, s_1, s_2\}$ here)
- An *initial state* $s_0 \in S$ (s_0 here)
- A set of *accepting* (or *final*) *states* $F \subset S$ ($\{s_2\}$ here)
- A *transition function* $\delta : S \times \Sigma \to 2^S$ (the arrows here)



[2]Note that an FA is *not* quite the same thing as a finite-state machine (FSM).

Finite Automata

Finite Automata

Notice the transition function δ outputs a *subset* of states.

Finite Automata

Notice the transition function δ outputs a *subset* of states.

In a *deterministic* finite automaton (DFA), the transition function always outputs a set with **exactly** one state (a *singleton*).

Finite Automata

Notice the transition function δ outputs a *subset* of states.

In a *deterministic* finite automaton (DFA), the transition function always outputs a set with **exactly** one state (a *singleton*).

i.e., in a DFA, the next state is **determined** by the input & current state.
(i.e., every state has exactly 1 arrow leaving it for each possible input.)

Finite Automata

Notice the transition function δ outputs a *subset* of states.

In a *deterministic* finite automaton (DFA), the transition function always outputs a set with **exactly** one state (a *singleton*).

i.e., in a DFA, the next state is **determined** by the input & current state.
(i.e., every state has exactly 1 arrow leaving it for each possible input.)

In a *nondeterministic* finite automaton (NFA), the above is not true.

Finite Automata

Finite automata are **language recognizers**: you feed a string as an input, and if it accepts the input string, the string is in its language.[3]

[3] *Pumping lemma*: A long-enough input must contain a repeatable substring. (Why?)

Finite Automata

Finite automata are **language recognizers**: you feed a string as an input, and if it accepts the input string, the string is in its language.[3]

In particular:
$\Longrightarrow$ Finite automata **recognize *regular* languages**, and *nothing else*!

[3] *Pumping lemma*: A long-enough input must contain a repeatable substring. (Why?)

Finite Automata

Finite automata are **language recognizers**: you feed a string as an input, and if it accepts the input string, the string is in its language.[3]

In particular:

$\Longrightarrow$ Finite automata **recognize *regular* languages**, and *nothing else*!

Therefore, we can:

[3] *Pumping lemma*: A long-enough input must contain a repeatable substring. (Why?)

Finite Automata

Finite automata are **language recognizers**: you feed a string as an input, and if it accepts the input string, the string is in its language.[3]

In particular:
$\Longrightarrow$ Finite automata **recognize *regular* languages**, and *nothing else*!

Therefore, we can:

1. Convert regex pattern to FA

[3] *Pumping lemma*: A long-enough input must contain a repeatable substring. (Why?)

Finite Automata

Finite automata are **language recognizers**: you feed a string as an input, and if it accepts the input string, the string is in its language.[3]

In particular:
$\implies$ Finite automata **recognize *regular* languages**, and *nothing else*!

Therefore, we can:

1. Convert regex pattern to FA
2. Feed corpus to FA

[3] *Pumping lemma*: A long-enough input must contain a repeatable substring. (Why?)

Finite Automata

Finite automata are **language recognizers**: you feed a string as an input, and if it accepts the input string, the string is in its language.[3]

In particular:
$\implies$ Finite automata **recognize *regular* languages**, and *nothing else*!

Therefore, we can:

1. Convert regex pattern to FA
2. Feed corpus to FA in **linear time**!

[3] *Pumping lemma*: A long-enough input must contain a repeatable substring. (Why?)

Finite Automata

Finite automata are **language recognizers**: you feed a string as an input, and if it accepts the input string, the string is in its language.[3]

In particular:
$\Longrightarrow$ Finite automata **recognize *regular* languages**, and *nothing else*!

Therefore, we can:

1. Convert regex pattern to FA
2. Feed corpus to FA in **linear time**!
3. ...

[3] *Pumping lemma*: A long-enough input must contain a repeatable substring. (Why?)

Finite Automata

Finite automata are **language recognizers**: you feed a string as an input, and if it accepts the input string, the string is in its language.[3]

In particular:
$\implies$ Finite automata **recognize *regular* languages**, and *nothing else*!

Therefore, we can:

1. Convert regex pattern to FA
2. Feed corpus to FA in **linear time**!
3. ...
4. Profit!

[3] *Pumping lemma*: A long-enough input must contain a repeatable substring. (Why?)

Finite Automata

Finite automata are **language recognizers**: you feed a string as an input, and if it accepts the input string, the string is in its language.[3]

In particular:
$\Longrightarrow$ Finite automata **recognize *regular* languages**, and *nothing else*!

Therefore, we can:

1. Convert regex pattern to FA
2. Feed corpus to FA in **linear time**!
3. ...
4. Profit!

But how can we do this?

[3] *Pumping lemma*: A long-enough input must contain a repeatable substring. (Why?)

Finite Automata from Regular Expressions

Consider: `(a|b)*(1+2|3)`.

Finite Automata from Regular Expressions

Consider: `(a|b)*(1+2|3)`. Ask: **Where in the pattern can we be?**

Finite Automata from Regular Expressions

Consider: `(a|b)*(1+2|3)`. Ask: **Where in the pattern can we be?**

1. $s_0 = \bullet$`(a|b)`*`(1+2|3)`

Finite Automata from Regular Expressions

Consider: `(a|b)*(1+2|3)`. Ask: **Where in the pattern can we be?**

1. $s_0 = \bullet$`(a|b)`*`(1+2|3)`
   $= \bullet$`(`$\bullet$`a|`$\bullet$`b)`*`(1+2|3)`

Finite Automata from Regular Expressions

Consider: `(a|b)*(1+2|3)`. Ask: **Where in the pattern can we be?**

1. $s_0 = \bullet\texttt{(a|b)}^*\texttt{(1+2|3)}$
   $= \bullet\texttt{(}\bullet\texttt{a|}\bullet\texttt{b)}^*\texttt{(1+2|3)}$
   $= \bullet\texttt{(}\bullet\texttt{a|}\bullet\texttt{b)}^*\bullet\texttt{(1+2|3)}$

Finite Automata from Regular Expressions

Consider: `(a|b)*(1+2|3)`. Ask: **Where in the pattern can we be?**

1. s_0 = •(a|b)*(1+2|3)
   = •(•a|•b)*(1+2|3)
   = •(•a|•b)*•(1+2|3)
   = •(•a|•b)*•(•1+2|•3)

Finite Automata from Regular Expressions

Consider: `(a|b)*(1+2|3)`. Ask: **Where in the pattern can we be?**

1. $s_0 = \bullet\texttt{(a|b)}^*\texttt{(1+2|3)}$
   $= \bullet\texttt{(}\bullet\texttt{a|}\bullet\texttt{b)}^*\texttt{(1+2|3)}$
   $= \bullet\texttt{(}\bullet\texttt{a|}\bullet\texttt{b)}^*\bullet\texttt{(1+2|3)}$
   $= \bullet\texttt{(}\bullet\texttt{a|}\bullet\texttt{b)}^*\bullet\texttt{(}\bullet\texttt{1+2|}\bullet\texttt{3)}$
2. $s_1 = \texttt{(a|b)}^*\texttt{(}\bullet\texttt{1+}\bullet\texttt{2|3)}$

Finite Automata from Regular Expressions

Consider: `(a|b)*(1+2|3)`. Ask: **Where in the pattern can we be?**

1. s_0 = `•(a|b)*(1+2|3)`
   = `•(•a|•b)*(1+2|3)`
   = `•(•a|•b)*•(1+2|3)`
   = `•(•a|•b)*•(•1+2|•3)`
2. s_1 = `(a|b)*(•1+•2|3)`
3. s_2 = `(a|b)*(1+2•|3)`

Finite Automata from Regular Expressions

Consider: `(a|b)*(1+2|3)`. Ask: **Where in the pattern can we be?**

1. $s_0 = \bullet$`(a|b)*(1+2|3)`
   $= \bullet$`(`$\bullet$`a|`$\bullet$`b)*(1+2|3)`
   $= \bullet$`(`$\bullet$`a|`$\bullet$`b)*`$\bullet$`(1+2|3)`
   $= \bullet$`(`$\bullet$`a|`$\bullet$`b)*`$\bullet$`(`$\bullet$`1+2|`$\bullet$`3)`
2. $s_1 =$ `(a|b)*(`$\bullet$`1+`$\bullet$`2|3)`
3. $s_2 =$ `(a|b)*(1+2`$\bullet$`|3)`
   $=$ `(a|b)*(1+2`$\bullet$`|3)`$\bullet$

Finite Automata from Regular Expressions

Consider: `(a|b)*(1+2|3)`. Ask: **Where in the pattern can we be?**

1. s_0 = •(a|b)*(1+2|3)
   = •(•a|•b)*(1+2|3)
   = •(•a|•b)*•(1+2|3)
   = •(•a|•b)*•(•1+2|•3)
2. s_1 = (a|b)*(•1+•2|3)
3. s_2 = (a|b)*(1+2•|3)
   = (a|b)*(1+2•|3)•

   s_2 = (a|b)*(1+2|3•)

Finite Automata from Regular Expressions

Consider: `(a|b)*(1+2|3)`. Ask: **Where in the pattern can we be?**

1. $$
\begin{aligned}
s_0 &= \bullet\texttt{(a|b)}^*\texttt{(1+2|3)} \\
&= \bullet\texttt{(}\bullet\texttt{a|}\bullet\texttt{b)}^*\texttt{(1+2|3)} \\
&= \bullet\texttt{(}\bullet\texttt{a|}\bullet\texttt{b)}^*\bullet\texttt{(1+2|3)} \\
&= \bullet\texttt{(}\bullet\texttt{a|}\bullet\texttt{b)}^*\bullet\texttt{(}\bullet\texttt{1+2|}\bullet\texttt{3)}
\end{aligned}
$$

2. $$s_1 = \texttt{(a|b)}^*\texttt{(}\bullet\texttt{1+}\bullet\texttt{2|3)}$$

3. $$
\begin{aligned}
s_2 &= \texttt{(a|b)}^*\texttt{(1+2}\bullet\texttt{|3)} \\
&= \texttt{(a|b)}^*\texttt{(1+2}\bullet\texttt{|3)}\bullet \\
s_2 &= \texttt{(a|b)}^*\texttt{(1+2|3}\bullet\texttt{)} \\
&= \texttt{(a|b)}^*\texttt{(1+2|3}\bullet\texttt{)}\bullet
\end{aligned}
$$

Finite Automata from Regular Expressions

Consider: `(a|b)*(1+2|3)`. Ask: **Where in the pattern can we be?**

1. s_0 = ●(a|b)*(1+2|3)
   = ●(●a|●b)*(1+2|3)
   = ●(●a|●b)*●(1+2|3)
   = ●(●a|●b)*●(●1+2|●3)

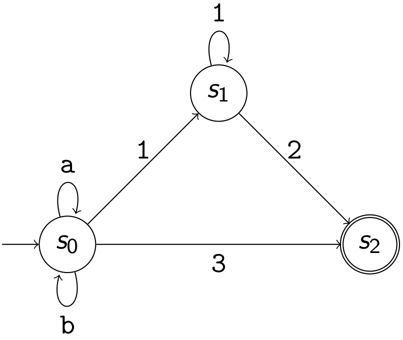
2. s_1 = (a|b)*(●1+●2|3)

3. s_2 = (a|b)*(1+2●|3)
   = (a|b)*(1+2●|3)●
   s_2 = (a|b)*(1+2|3●)
   = (a|b)*(1+2|3●)●
   s_2 = (a|b)*(1+2●|3●)●●

Finite Automata from Regular Expressions

Consider: `(a|b)*(1+2|3)`. Ask: **Where in the pattern can we be?**

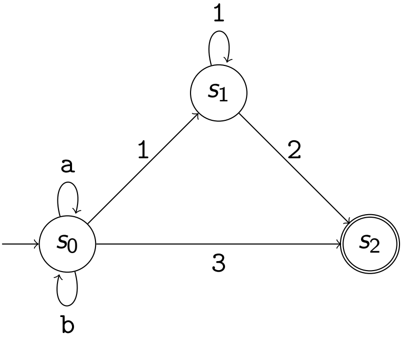
1. s_0 = •(a|b)*(1+2|3)
   = •(•a|•b)*(1+2|3)
   = •(•a|•b)*•(1+2|3)
   = •(•a|•b)*•(•1+2|•3)
2. s_1 = (a|b)*(•1+•2|3)
3. s_2 = (a|b)*(1+2•|3)
   = (a|b)*(1+2•|3)•
   s_2 = (a|b)*(1+2|3•)
   = (a|b)*(1+2|3•)•
   s_2 = (a|b)*(1+2•|3•)••



Finite Automata from Regular Expressions

Consider: (a|b)*(1+2|3). Ask: **Where in the pattern can we be?**

1. s_0 = •(a|b)*(1+2|3)
   = •(•a|•b)*(1+2|3)
   = •(•a|•b)*•(1+2|3)
   = •(•a|•b)*•(•1+2|•3)
2. s_1 = (a|b)*(•1+•2|3)
3. s_2 = (a|b)*(1+2•|3)
   = (a|b)*(1+2•|3)•
   s_2 = (a|b)*(1+2|3•)
   = (a|b)*(1+2|3•)•
   s_2 = (a|b)*(1+2•|3•)••



(Expanding a state to its equivalents is a mathematical *closure* operation.)

Finite Automata from Regular Expressions

Finite Automata from Regular Expressions

We created a *deterministic finite automaton* (DFA) from a regex!

Finite Automata from Regular Expressions

We created a *deterministic finite automaton* (DFA) from a regex!

It can find regular patterns (substrings, subsequences, etc.) in **linear** time.

Finite Automata from Regular Expressions

We created a *deterministic finite automaton* (DFA) from a regex!

It can find regular patterns (substrings, subsequences, etc.) in **linear** time.

However: *there is no such thing as a free lunch.*

Finite Automata from Regular Expressions

We created a *deterministic finite automaton* (DFA) from a regex!

It can find regular patterns (substrings, subsequences, etc.) in **linear** time.

However: *there is no such thing as a free lunch.*

What is the caveat?

Finite Automata from Regular Expressions

Caveat:

Finite Automata from Regular Expressions

Caveat:

The number of states $|S|$ can be **exponential** in the size of the pattern m.

Finite Automata from Regular Expressions

Caveat:

The number of states $|S|$ can be **exponential** in the size of the pattern m. (This is sometimes referred to as the size of the *state space*.)

Caveat:

The number of states $|S|$ can be **exponential** in the size of the pattern m. (This is sometimes referred to as the size of the *state space*.)

Why?

Finite Automata from Regular Expressions

Caveat:

The number of states $|S|$ can be **exponential** in the size of the pattern m. (This is sometimes referred to as the size of the *state space*.)

Why? Because we compute **subsets** of locations in the pattern, and we could encounter around 2^m subsets for a pattern of length m.

Caveat:

The number of states $|S|$ can be **exponential** in the size of the pattern m. (This is sometimes referred to as the size of the *state space*.)

Why? Because we compute **subsets** of locations in the pattern, and we could encounter around 2^m subsets for a pattern of length m.

Solution?

Finite Automata from Regular Expressions

Caveat:

The number of states $|S|$ can be **exponential** in the size of the pattern m. (This is sometimes referred to as the size of the *state space*.)

Why? Because we compute **subsets** of locations in the pattern, and we could encounter around 2^m subsets for a pattern of length m.

Solution?

DFA minimization: Always merge states that behave identically.

Finite Automata from Regular Expressions

Caveat:

The number of states $|S|$ can be **exponential** in the size of the pattern m. (This is sometimes referred to as the size of the *state space*.)

Why? Because we compute **subsets** of locations in the pattern, and we could encounter around 2^m subsets for a pattern of length m.

Solution?

DFA minimization: Always merge states that behave identically.
$\rightarrow$ We already did this. It often works well.

Finite Automata from Regular Expressions

Caveat:

The number of states $|S|$ can be **exponential** in the size of the pattern m. (This is sometimes referred to as the size of the *state space*.)

Why? Because we compute **subsets** of locations in the pattern, and we could encounter around 2^m subsets for a pattern of length m.

Solution?

DFA minimization: Always merge states that behave identically.
$\rightarrow$ We already did this. It often works well.

Boring: Use an NFA instead of a DFA. (Track state items separately.)

Finite Automata from Regular Expressions

Caveat:

The number of states $|S|$ can be **exponential** in the size of the pattern m. (This is sometimes referred to as the size of the *state space*.)

Why? Because we compute **subsets** of locations in the pattern, and we could encounter around 2^m subsets for a pattern of length m.

Solution?

DFA minimization: Always merge states that behave identically.
$\rightarrow$ We already did this. It often works well.

Boring: Use an NFA instead of a DFA. (Track state items separately.)
$\rightarrow$ Guarantees $O(m)$ memory usage, but running time is $O(mn)$.

Finite Automata from Regular Expressions

Caveat:

The number of states $|S|$ can be **exponential** in the size of the pattern m. (This is sometimes referred to as the size of the *state space*.)

Why? Because we compute **subsets** of locations in the pattern, and we could encounter around 2^m subsets for a pattern of length m.

Solution?

DFA minimization: Always merge states that behave identically.
$\rightarrow$ We already did this. It often works well.

Boring: Use an NFA instead of a DFA. (Track state items separately.)
$\rightarrow$ Guarantees $O(m)$ memory usage, but running time is $O(mn)$.

Clever: Build the DFA lazily as needed by input.

Finite Automata from Regular Expressions

Caveat:

The number of states $|S|$ can be **exponential** in the size of the pattern m. (This is sometimes referred to as the size of the *state space*.)

Why? Because we compute **subsets** of locations in the pattern, and we could encounter around 2^m subsets for a pattern of length m.

Solution?

DFA minimization: Always merge states that behave identically.
$\rightarrow$ We already did this. It often works well.

Boring: Use an NFA instead of a DFA. (Track state items separately.)
$\rightarrow$ Guarantees $O(m)$ memory usage, but running time is $O(mn)$.

Clever: Build the DFA lazily as needed by input.
$\rightarrow$ Memory usage becomes $O(m + n)$, but running time approaches $O(n)$.

Conclusion

Conclusion

Automata are extremely powerful!

Conclusion

Automata are extremely powerful! They can do many other cool things:

Conclusion

Automata are extremely powerful! They can do many other cool things:

- *Levenshtein automata* can recognize corpora that are k "edit distances" (insertions, deletions, or mutations) away from a pattern.

Conclusion

Automata are extremely powerful! They can do many other cool things:

- *Levenshtein automata* can recognize corpora that are k "edit distances" (insertions, deletions, or mutations) away from a pattern.
- When given a stack, LR automata can parse *context-free languages* (like many programming languages) in linear time (CS 164).

Conclusion

Automata are extremely powerful! They can do many other cool things:

- *Levenshtein automata* can recognize corpora that are k "edit distances" (insertions, deletions, or mutations) away from a pattern.
- When given a stack, LR automata can parse *context-free languages* (like many programming languages) in linear time (CS 164).
- Büchi automata, which allow *infinite-length* input strings, are used for formal verification of computer programs.

Conclusion

Automata are extremely powerful! They can do many other cool things:

- *Levenshtein automata* can recognize corpora that are k "edit distances" (insertions, deletions, or mutations) away from a pattern.
- When given a stack, LR automata can parse *context-free languages* (like many programming languages) in linear time (CS 164).
- Büchi automata, which allow *infinite-length* input strings, are used for formal verification of computer programs.

Finite-state machines (very similar) are widely used in digital design:

Conclusion

Automata are extremely powerful! They can do many other cool things:

- *Levenshtein automata* can recognize corpora that are k "edit distances" (insertions, deletions, or mutations) away from a pattern.
- When given a stack, LR automata can parse *context-free languages* (like many programming languages) in linear time (CS 164).
- Büchi automata, which allow *infinite-length* input strings, are used for formal verification of computer programs.

Finite-state machines (very similar) are widely used in digital design:

- Used in engineering to prove digital systems work as intended

Conclusion

Automata are extremely powerful! They can do many other cool things:

- *Levenshtein automata* can recognize corpora that are k "edit distances" (insertions, deletions, or mutations) away from a pattern.
- When given a stack, LR automata can parse *context-free languages* (like many programming languages) in linear time (CS 164).
- Büchi automata, which allow *infinite-length* input strings, are used for formal verification of computer programs.

Finite-state machines (very similar) are widely used in digital design:

- Used in engineering to prove digital systems work as intended
- Used to optimize power consumption, logic circuitry, etc.

Conclusion

Conclusion

This is just the tip of the iceberg for string algorithms (and automata). Languages, grammars, and automata are also used in computational linguistics, computational biology/genomics (DNA alignment/matching)...

Conclusion

This is just the tip of the iceberg for string algorithms (and automata).
Languages, grammars, and automata are also used in computational linguistics, computational biology/genomics (DNA alignment/matching)...

It is extremely easy to graduate and avoid languages & automata.
But they provide the keys for solving many otherwise difficult problems.
You can see more in EE/CS 144, 149, 151, 164, 172, 176, 219C, 291E...

Conclusion

This is just the tip of the iceberg for string algorithms (and automata).
Languages, grammars, and automata are also used in computational linguistics, computational biology/genomics (DNA alignment/matching)...

It is extremely easy to graduate and avoid languages & automata.
But they provide the keys for solving many otherwise difficult problems.
You can see more in EE/CS 144, 149, 151, 164, 172, 176, 219C, 291E...

$\sim$ **Related Words of Wisdom** $\sim$

Conclusion

This is just the tip of the iceberg for string algorithms (and automata). Languages, grammars, and automata are also used in computational linguistics, computational biology/genomics (DNA alignment/matching)...

It is extremely easy to graduate and avoid languages & automata. But they provide the keys for solving many otherwise difficult problems. You can see more in EE/CS 144, 149, 151, 164, 172, 176, 219C, 291E...

$\sim$ **Related Words of Wisdom** $\sim$

Minimizing the number of states in your design (e.g. factoring out duplicate data) helps keep designs clean & bug-free.

Conclusion

This is just the tip of the iceberg for string algorithms (and automata).
Languages, grammars, and automata are also used in computational linguistics, computational biology/genomics (DNA alignment/matching)...

It is extremely easy to graduate and avoid languages & automata.
But they provide the keys for solving many otherwise difficult problems.
You can see more in EE/CS 144, 149, 151, 164, 172, 176, 219C, 291E...

$\sim$ **Related Words of Wisdom** $\sim$

Minimizing the number of states in your design (e.g. factoring out duplicate data) helps keep designs clean & bug-free.

One reason: single source of truth. If it *can't* be wrong, it won't be.

Conclusion

This is just the tip of the iceberg for string algorithms (and automata).
Languages, grammars, and automata are also used in computational linguistics, computational biology/genomics (DNA alignment/matching)...

It is extremely easy to graduate and avoid languages & automata.
But they provide the keys for solving many otherwise difficult problems.
You can see more in EE/CS 144, 149, 151, 164, 172, 176, 219C, 291E...

∼ Related Words of Wisdom ∼

Minimizing the number of states in your design (e.g. factoring out duplicate data) helps keep designs clean & bug-free.

One reason: single source of truth. If it *can't* be wrong, it won't be.

Kleeneliness is next to Gödeliness.

...Calculus?

...Calculus?

Bonus: What language does `S` describe?

$$\texttt{S} \rightarrow \texttt{S a} \mid \varepsilon$$

...Calculus?

Bonus: What language does S describe?

$$\texttt{S} \rightarrow \texttt{S a} \mid \varepsilon$$

Hmm, union and concatenation sure look like addition & multiplication...

...Calculus?

Bonus: What language does S describe?

$$\texttt{S} \rightarrow \texttt{S a} \mid \varepsilon$$

Hmm, union and concatenation sure look like addition & multiplication...

$$S = Sa + \varepsilon$$

...Calculus?

Bonus: What language does `S` describe?

$$\texttt{S} \rightarrow \texttt{S a} \mid \varepsilon$$

Hmm, union and concatenation sure look like addition & multiplication...

$$S = Sa + \varepsilon$$
$$S\varepsilon = Sa + \varepsilon$$

...Calculus?

Bonus: What language does `S` describe?

$$\texttt{S} \rightarrow \texttt{S a} \mid \varepsilon$$

Hmm, union and concatenation sure look like addition & multiplication...

$$S = Sa + \varepsilon$$
$$S\varepsilon = Sa + \varepsilon$$
$$S(\varepsilon - a) = \varepsilon$$

...Calculus?

Bonus: What language does S describe?

$$\texttt{S} \rightarrow \texttt{S a} \mid \varepsilon$$

Hmm, union and concatenation sure look like addition & multiplication...

$$\begin{aligned} S &= Sa + \varepsilon \\ S\varepsilon &= Sa + \varepsilon \\ S(\varepsilon - a) &= \varepsilon \\ S &= \frac{\varepsilon}{\varepsilon - a} \end{aligned}$$

...Calculus?

Bonus: What language does S describe?

$$\texttt{S} \rightarrow \texttt{S a} \mid \varepsilon$$

Hmm, union and concatenation sure look like addition & multiplication...

$$S = Sa + \varepsilon$$
$$S\varepsilon = Sa + \varepsilon$$
$$S(\varepsilon - a) = \varepsilon$$
$$S = \frac{\varepsilon}{\varepsilon - a}$$

...wait, what?

...Calculus?

Bonus: What language does `S` describe?

$$\texttt{S} \rightarrow \texttt{S a} \mid \varepsilon$$

Hmm, union and concatenation sure look like addition & multiplication...

$$
\begin{aligned}
S &= Sa + \varepsilon \\
S\varepsilon &= Sa + \varepsilon \\
S(\varepsilon - a) &= \varepsilon \\
S &= \frac{\varepsilon}{\varepsilon - a}
\end{aligned}
$$

...wait, what?

Oh, right—Taylor series...

...Calculus?

Bonus: What language does `S` describe?

$$\texttt{S} \rightarrow \texttt{S a} \mid \varepsilon$$

Hmm, union and concatenation sure look like addition & multiplication...

$$S = Sa + \varepsilon$$
$$S\varepsilon = Sa + \varepsilon$$
$$S(\varepsilon - a) = \varepsilon$$
$$S = \frac{\varepsilon}{\varepsilon - a}$$

...wait, what?

Oh, right—Taylor series...

$$S = \varepsilon + a + a^2 + \ldots$$

...Calculus?

Bonus: What language does `S` describe?

$$\texttt{S} \rightarrow \texttt{S a} \mid \varepsilon$$

Hmm, union and concatenation sure look like addition & multiplication...

$$S = Sa + \varepsilon$$
$$S\varepsilon = Sa + \varepsilon$$
$$S(\varepsilon - a) = \varepsilon$$
$$S = \frac{\varepsilon}{\varepsilon - a}$$

...wait, what?

Oh, right—Taylor series...

$$S = \varepsilon + a + a^2 + \ldots$$

:-) Please don't ask...

Conclusion

Thank you!