Regular Expressions
Recognizers vs Generators

We want to write expressions that generate strings.
We want to write expressions that generate strings.
Regular Expressions over $\Sigma$

All the cool kids call them regex.

Def
All the cool kids call them regex.

**Def**

1. $e$ is a regex. Every $\sigma \in \Sigma$ is a regex.
Regular Expressions over $\Sigma$

All the cool kids call them regex.

**Def**

1. $e$ is a regex. Every $\sigma \in \Sigma$ is a regex.
2. If $\alpha$ and $\beta$ are regex then $\alpha \cup \beta$ and $\alpha\beta$ are regex.
Regular Expressions over $\Sigma$

All the cool kids call them regex.

Def

1. $e$ is a regex. Every $\sigma \in \Sigma$ is a regex.
2. If $\alpha$ and $\beta$ are regex then $\alpha \cup \beta$ and $\alpha \beta$ are regex.
3. If $\alpha$ is a regex then $\alpha^*$ is a regex.
All the cool kids call them regex.

**Def**

1. $e$ is a regex. Every $\sigma \in \Sigma$ is a regex.
2. If $\alpha$ and $\beta$ are regex then $\alpha \cup \beta$ and $\alpha \beta$ are regex.
3. If $\alpha$ is a regex then $\alpha^*$ is a regex.

Need to give examples and assign meaning.
Example and Meaning

A regex represents a set
Example and Meaning

A regex represents a set

\( a \) is a regex. It represents \( \{ a \} \).
A regex represents a set

\( a \) is a regex. It represents \( \{a\} \).

\( a^* \) is a regex. It represents \( \{e, a, aa, aaa, \ldots\} \).
A regex represents a set

- $a$ is a regex. It represents $\{a\}$.
- $a^*$ is a regex. It represents $\{e, a, aa, aaa, \ldots\}$.
- $a^*b$ is a regex. It represents $\{b, ab, aab, aaab, \ldots\}$.
Example and Meaning

A regex represents a set

- $a$ is a regex. It represents $\{a\}$.
- $a^*$ is a regex. It represents $\{e, a, aa, aaa, \ldots\}$.
- $a^*b$ is a regex. It represents $\{b, ab, aab, aaab, \ldots\}$.
- $a^*b \cup b^*$ is a regex. You can guess what it represents.
A regex represents a set

- \(a\) is a regex. It represents \(\{a\}\).
- \(a^*\) is a regex. It represents \(\{e, a, aa, aaa, \ldots\}\).
- \(a^*b\) is a regex. It represents \(\{b, ab, aab, aaab, \ldots\}\).
- \(a^*b \cup b^*\) is a regex. You can guess what it represents.

**Def** If \(\alpha\) is a regex then \(L(\alpha)\) is the set of strings it generates.
Examples

1. $b^*(ab*ab*)*ab^*$
2. $b^*(ab*ab*ab*)^*$
3. $(b^*(ab*ab*)*ab^*) \cup (b^*(ab*ab*ab*)^*)$
\[ L((b^*(ab^*ab^*)^*) ab^*) \cup (b^*(ab^*ab^*ab^*)^*) \] is accepted by an NFA)
How is Regex related to Regular?

A language generated by a regular expression if and only if it is recognized by a finite automaton.

We know:
- DFA is equivalent to NFA

Will show:
- Lemma: If a language is generated by a regular expression, it is recognized by an NFA.
- Lemma: If a language is recognized by a DFA, it is generated by a regular expression.

QED
How is Regex related to Regular?

**Thm** A language generated by a regular expression if and only if it is recognized by a finite automaton.
How is Regex related to Regular?

**Thm** A language generated by a regular expression if and only if it is recognized by a finite automaton.

**Pf**
How is Regex related to Regular?

**Thm** A language generated by a regular expression if and only if it is recognized by a finite automaton.

**Pf**

We know:

DFA is equivalent to NFA
How is Regex related to Regular?

**Thm** A language generated by a regular expression if and only if it is recognized by a finite automaton.

**Pf**

We know:

- DFA is equivalent to NFA

**WEAK**

**STRONG**
**Thm** A language generated by a regular expression if and only if it is recognized by a finite automaton.

**Pf**

We know:

- DFA is equivalent to NFA

**WEAK** **STRONG**

Will show:
**Thm** A language generated by a regular expression if and only if it is is recognized by a finite automaton.

**Pf**

We know:

- DFA is equivalent to NFA

**Lemma** If a language is generated by a regular expression, it is recognized by an NFA.
How is Regex related to Regular?

**Thm** A language generated by a regular expression if and only if it is is recognized by a finite automaton.

**Pf**

We know:

DFA is equivalent to NFA

**WEAK**

**STRONG**

Will show:

**Lemma** If a language is generated by a regular expression, it is recognized by an NFA.

**Lemma** If a language is recognized by a DFA, it is generated by a regular expression.
How is Regex related to Regular?

**Thm** A language generated by a regular expression if and only if it is is recognized by a finite automaton.

**Pf**

We know:

DFA is equivalent to NFA

**WEAK STRONG**

Will show:

**Lemma** If a language is generated by a regular expression, it is recognized by an NFA.

**Lemma** If a language is recognized by a DFA, it is generated by a regular expression.

**QED**
Lemma If a language is generated by a regular expression, it is recognized by an NFA.
Lemma If a language is generated by a regular expression, it is recognized by an NFA.
Pf By **structural induction** on the formation of a regex
Lemma If a language is generated by a regular expression, it is recognized by an NFA.

Pf By **structural induction** on the formation of a regex (... or by **strong induction** on the length).
Lemma If a language is generated by a regular expression, it is recognized by an NFA.

Pf By **structural induction** on the formation of a regex (... or by **strong induction** on the length).

Base Cases $e$ and $\{\sigma\}$ have NFA’s.
Lemma If a language is generated by a regular expression, it is recognized by an NFA.

Pf By structural induction on the formation of a regex (... or by strong induction on the length).

Base Cases $e$ and $\{\sigma\}$ have NFA’s.
Lemma If a language is generated by a regular expression, it is recognized by an NFA.

Pf By **structural induction** on the formation of a regex (... or by **strong induction** on the length).

**Base Cases** $e$ and $\{\sigma\}$ have NFA’s.
Lemma If a language is generated by a regular expression, it is recognized by an NFA.

Pf By structural induction on the formation of a regex (... or by strong induction on the length).

Base Cases $e$ and $\{\sigma\}$ have NFA’s.

IS Let $\alpha$ be a regex.
Lemma If a language is generated by a regular expression, it is recognized by an NFA.

Pf By structural induction on the formation of a regex (... or by strong induction on the length).

Base Cases $e$ and $\{\sigma\}$ have NFA’s.

IS Let $\alpha$ be a regex.

Case 1 $\alpha = \alpha_1 \cup \alpha_2$. Since $|\alpha_1| < n$, $|\alpha_2| < n$, apply IH: NFA’s $N_i$ for $\alpha_i$. Use closure of NFAs under union to get NFA for $L(N_1) \cup L(N_2)$. This is NFA for $L(\alpha)$.
Lemma If a language is generated by a regular expression, it is recognized by an NFA.

Pf By structural induction on the formation of a regex (... or by strong induction on the length).

Base Cases $e$ and $\{\sigma\}$ have NFA’s.

IS Let $\alpha$ be a regex.

Case 1 $\alpha = \alpha_1 \cup \alpha_2$. Since $|\alpha_1| < n$, $|\alpha_2| < n$, apply IH: NFA’s $N_i$ for $\alpha_i$. Use closure of NFAs under union to get NFA for $L(N_1) \cup L(N_2)$. This is NFA for $L(\alpha)$.

Case 2 $\alpha = \alpha_1 \cdot \alpha_2$. Similar. Use closure under concatenation.
Lemma If a language is generated by a regular expression, it is recognized by an NFA.

Pf By structural induction on the formation of a regex (... or by strong induction on the length).

Base Cases $e$ and $\{\sigma\}$ have NFA’s.

IS Let $\alpha$ be a regex.

Case 1 $\alpha = \alpha_1 \cup \alpha_2$. Since $|\alpha_1| < n$, $|\alpha_2| < n$, apply IH: NFA’s $N_i$ for $\alpha_i$. Use closure of NFAs under union to get NFA for $L(N_1) \cup L(N_2)$. This is NFA for $L(\alpha)$.

Case 2 $\alpha = \alpha_1 \cdot \alpha_2$. Similar. Use closure under concatenation.

Case 3 $\alpha = \alpha_1^*$. Similar. Use closure under Kleene *. 
If $\alpha$ was of length $n$ then the NFA you get for it has $\leq 2n$ states.
Lemma If a language is recognized by a DFA, it is generated by a regular expression.
Lemma If a language is recognized by a DFA, it is generated by a regular expression.

Pf Assume DFA has start state \( s \) and final states \( f_1, \ldots, f_m \).
Lemma If a language is recognized by a DFA, it is generated by a regular expression.

Pf Assume DFA has start state $s$ and final states $f_1, \ldots, f_m$. For each $f_i$, we will produce a regex, $E(s, f_i)$, that generates all words recognized by starting in $s$ and ending in final state $f_i$. Then the desired regex is $E(s, f_1) \cup E(s, f_2) \cup \cdots \cup E(s, f_m)$. 
Lemma If a language is recognized by a DFA, it is generated by a regular expression.

Pf Assume DFA has start state $s$ and final states $f_1, \ldots, f_m$. For each $f_i$, we will produce a regex, $E(s, f_i)$, that generates all words recognized by starting in $s$ and ending in final state $f_i$. Then the desired regex is

$$E(s, f_1) \cup E(s, f_2) \cup \cdots \cup E(s, f_m)$$
Notation: $\delta(q, w)$

Given a DFA $M = (Q, \Sigma, \delta, s, F)$ we note that

$$\delta: Q \times \Sigma \rightarrow Q.$$
Given a DFA $M = (Q, \Sigma, \delta, s, F)$ we note that

$$\delta : Q \times \Sigma \rightarrow Q.$$ 

We can extend $\delta$ to strings

$$\delta : Q \times \Sigma^* \rightarrow Q.$$
Notation: $\delta(q, w)$

Given a DFA $M = (Q, \Sigma, \delta, s, F)$ we note that

$$\delta : Q \times \Sigma \rightarrow Q.$$  

We can extend $\delta$ to strings

$$\delta : Q \times \Sigma^* \rightarrow Q.$$  

$\delta(q, w) =$ State that $M$ ends up in if start at $q$ and feed in the string
Notation: $\delta(q, w)$

Given a DFA $M = (Q, \Sigma, \delta, s, F)$ we note that

$$\delta : Q \times \Sigma \rightarrow Q.$$  

We can extend $\delta$ to strings

$$\delta : Q \times \Sigma^* \rightarrow Q.$$  

$\delta(q, w) =$ State that $M$ ends up in if start at $q$ and feed in the string $w$.

What about the empty string?

$\delta(q, \epsilon) =$ State that $M$ ends up in if start at $q$ and feed in the empty string.
Notation: \( \delta(q, w) \)

Given a DFA \( M = (Q, \Sigma, \delta, s, F) \) we note that

\[
\delta : Q \times \Sigma \to Q.
\]

We can extend \( \delta \) to strings

\[
\delta : Q \times \Sigma^* \to Q.
\]

\( \delta(q, w) = \) State that \( M \) ends up in if start at \( q \) and feed in the string \( w \).

What about the empty string?

\[
\delta(q, e) = q.
\]
Given a DFA $M$ we want a Regex for $L(M)$. 
Given a DFA $M$ we want a Regex for $L(M)$.

**Key** We will find, for every pair of states $(i, j)$ the regex that represents strings that take you from state $i$ to state $j$. 
Given a DFA $M$ we want a Regex for $L(M)$.

**Key** We will find, for every pair of states $(i,j)$ the regex that represents strings that take you from state $i$ to state $j$.

**Why?** That seems like way more than we need.
**DFA ⊆ REGEX**

Given a DFA $M$ we want a Regex for $L(M)$.

**Key** We will find, for every pair of states $(i, j)$ the regex that represents strings that take you from state $i$ to state $j$.

**Why?** That seems like way more than we need.

**Dynamic Programming** We will use all of this information to get our final answer.
Definition of $R(i, j, k)$

Will assume $M$ has state set $\{1, \ldots, n\}$. I wrote on the last slide:
Definition of $R(i, j, k)$

Will assume $M$ has state set $\{1, \ldots, n\}$. 

I wrote on the last slide:

**Key** We will find, for every pair of states $(i, j)$ the regex that represents strings that take you from state $i$ to state $j$. 

For all $1 \leq i, j \leq n$ and $0 \leq k \leq n$, we will find a regex for $R(i, j, k)$. 


Definition of $R(i, j, k)$

Will assume $M$ has state set $\{1, \ldots, n\}$.
I wrote on the last slide:

**Key** We will find, for every pair of states $(i, j)$ the regex that represents strings that take you from state $i$ to state $j$.
Actually we will find out a lot more information.
Will assume $M$ has state set $\{1, \ldots, n\}$.
Definition of $R(i, j, k)$

Will assume $M$ has state set $\{1, \ldots, n\}$. I wrote on the last slide:

**Key** We will find, for every pair of states $(i, j)$ the regex that represents strings that take you from state $i$ to state $j$.

Actually we will find out a lot more information. Will assume $M$ has state set $\{1, \ldots, n\}$.

$$R(i, j, k) = \{w : \delta(i, w) = j \text{ but only use states in } \{1, \ldots, k\} \}.$$
Definition of $R(i, j, k)$

Will assume $M$ has state set $\{1, \ldots, n\}$.

I wrote on the last slide:

**Key** We will find, for every pair of states $(i, j)$ the regex that represents strings that take you from state $i$ to state $j$.

Actually we will find out a lot more information.

Will assume $M$ has state set $\{1, \ldots, n\}$.

\[
R(i, j, k) = \{ w : \delta(i, w) = j \text{ but only use states in } \{1, \ldots, k\} \}.
\]

For all $1 \leq i, j \leq n$ $0 \leq k \leq n$, we will find a regex for $R(i, j, k)$.
Finding Regex for $R(i, j, k)$

\[ R(i, j, k) = \{ w : \delta(i, w) = j \text{ but only use states in } \{1, \ldots, k\} \}. \]
Finding Regex for $R(i, j, k)$

$R(i, j, k) = \{ w : \delta(i, w) = j \text{ but only use states in } \{1, \ldots, k\} \}$. 

We will first find Regex for $R(i, j, 0)$ for all $1 \leq i, j \leq n$. 
Finding Regex for $R(i, j, k)$

$R(i, j, k) = \{ w : \delta(i, w) = j \text{ but only use states in } \{1, \ldots, k\} \}$. 

We will first find Regex for $R(i, j, 0)$ for all $1 \leq i, j \leq n$. 

What is $R(i, j, 0)$? 

If a string goes from $i$ to $j$ with no intermediary states then it must just be a transition.
Finding Regex for $R(i, j, k)$

$$R(i, j, k) = \{ w : \delta(i, w) = j \text{ but only use states in } \{1, \ldots, k\} \}.$$  

We will first find Regex for $R(i, j, 0)$ for all $1 \leq i, j \leq n$.

What is $R(i, j, 0)$?

If a string goes from $i$ to $j$ with no intermediary states then it must just be a transition. 

Or $i = j$ and the string that is $e$. 

Finding Regex for $R(i, j, k)$

$$R(i, j, k) = \{ w : \delta(i, w) = j \text{ but only use states in } \{1, \ldots, k\} \}.$$  

We will first find Regex for $R(i, j, 0)$ for all $1 \leq i, j \leq n$.  

What is $R(i, j, 0)$?  

If a string goes from $i$ to $j$ with no intermediary states then it must just be a transition.  

Or $i = j$ and the string that is $e$.

$$R(i, j, 0) = \begin{cases}  
\{ \sigma : \delta(i, \sigma) = j \} & \text{if } i \neq j \\  
\{ \sigma : \delta(i, \sigma) = j \} \cup \{ e \} & \text{if } i = j 
\end{cases} \quad (1)$$
$R(i, j, 0)$ is a Regex. Inductive Step

\[
R(i, j, 0) = \begin{cases} 
\{ \sigma : \delta(i, \sigma) = j \} & \text{if } i \neq j \\ 
\{ \sigma : \delta(i, \sigma) = j \} \cup \{ e \} & \text{if } i = j 
\end{cases}
\]  

(2)
\( R(i, j, 0) \) is a Regex. Inductive Step

\[
R(i, j, 0) = \begin{cases} 
\{ \sigma : \delta(i, \sigma) = j \} & \text{if } i \neq j \\
\{ \sigma : \delta(i, \sigma) = j \} \cup \{ e \} & \text{if } i = j 
\end{cases}
\]  \hspace{1cm} (2)

In both cases \( R(i, j, 0) \) can be expressed as a Regex.
\( R(i, j, 0) \) is a Regex. Inductive Step

\[
R(i, j, 0) = \begin{cases} 
\{ \sigma : \delta(i, \sigma) = j \} & \text{if } i \neq j \\
\{ \sigma : \delta(i, \sigma) = j \} \cup \{ e \} & \text{if } i = j 
\end{cases}
\]  

(2)

In both cases \( R(i, j, 0) \) can be expressed as a Regex.

We will now assume that for all \( 1 \leq i, j \leq n, R(i, j, k - 1) \) is a Regex and prove that for all \( 1 \leq i, j \leq n, R(i, j, k) \) is a Regex.
$R(i, j, 0)$ is a Regex. Inductive Step

$$R(i, j, 0) = \begin{cases} 
\{ \sigma : \delta(i, \sigma) = j \} & \text{if } i \neq j \\
\{ \sigma : \delta(i, \sigma) = j \} \cup \{ e \} & \text{if } i = j
\end{cases} \quad (2)$$

In both cases $R(i, j, 0)$ can be expressed as a Regex.

We will now **assume** that for all $1 \leq i, j \leq n$, $R(i, j, k - 1)$ is a Regex and **prove** that for all $1 \leq i, j \leq n$, $R(i, j, k)$ is a Regex.

This is both of the following:
$R(i, j, 0)$ is a Regex. Inductive Step

$$R(i, j, 0) = \begin{cases} \{ \sigma : \delta(i, \sigma) = j \} & \text{if } i \neq j \\ \{ \sigma : \delta(i, \sigma) = j \} \cup \{e\} & \text{if } i = j \end{cases} \quad (2)$$

In both cases $R(i, j, 0)$ can be expressed as a Regex.

We will now assume that for all $1 \leq i, j \leq n$, $R(i, j, k - 1)$ is a Regex and prove that for all $1 \leq i, j \leq n$, $R(i, j, k)$ is a Regex.

This is both of the following:

1. A proof by induction on $k$ that, for all $1 \leq i, j \leq n$, $R(i, j, k)$ is a Regex.
2. A dynamic program that computes all $R(i, j, k)$. 
Inductive Step $R(i, j, k)$ as a Picture

\[ R(i, k, k - 1) \rightarrow R(k, j, k - 1) \]
Complete Proof on One Slide

For all $1 \leq i, j \leq n$:

$$R(i,j,0) = \begin{cases} 
\{ \sigma : \delta(i, \sigma) = j \} & \text{if } i \neq j \\
\{ \sigma : \delta(i, \sigma) = j \} \cup \{ e \} & \text{if } i = j 
\end{cases}$$ (3)

All $R(i,j,0)$ are Regex.
Complete Proof on One Slide

For all $1 \leq i, j \leq n$:

$$R(i, j, 0) = \begin{cases} 
\{ \sigma : \delta(i, \sigma) = j \} & \text{if } i \neq j \\
\{ \sigma : \delta(i, \sigma) = j \} \cup \{ e \} & \text{if } i = j 
\end{cases}$$ (3)

All $R(i, j, 0)$ are Regex.
For all $1 \leq i, j \leq n$:

$$R(i, j, 0) = \begin{cases} \
\{ \sigma : \delta(i, \sigma) = j \} & \text{if } i \neq j \\
\{ \sigma : \delta(i, \sigma) = j \} \cup \{ e \} & \text{if } i = j \end{cases}$$ \hspace{1cm} (3)$$

All $R(i, j, 0)$ are Regex.

For all $1 \leq i, j \leq n$ and all $0 \leq k \leq n$

$$R(i, j, k) = R(i, j, k-1) \bigcup R(i, k, k-1)R(k, k, k-1)^{*}R(k, j, k-1)$$
Complete Proof on One Slide

For all $1 \leq i, j \leq n$:

$$R(i, j, 0) = \begin{cases} \{ \sigma : \delta(i, \sigma) = j \} & \text{if } i \neq j \\ \{ \sigma : \delta(i, \sigma) = j \} \cup \{ e \} & \text{if } i = j \end{cases} \quad (3)$$

All $R(i, j, 0)$ are Regex.

For all $1 \leq i, j \leq n$ and all $0 \leq k \leq n$

$$R(i, j, k) = R(i, j, k-1) \cup R(i, k, k-1)R(k, k, k-1)^*R(k, j, k-1)$$

If ALL $R(i, j, k - 1)$ are Regex, then ALL $R(i, j, k)$ are Regex.
Textbook Regular Expressions

Recall that \( \{a, b\}^*a\{a, b\}^n \).

1. DFA requires \( 2^{n+1} \) states.
2. NFA can be done with \( n + 2 \) states.
3. How long is the regex for it? Regard the \( \{a, b\}^*a \) part to be \( O(1) \) length.
Recall that \( \{a, b\}^* a \{a, b\}^n \).

1. DFA requires \( 2^{n+1} \) states.
2. NFA can be done with \( n + 2 \) states.
3. How long is the regex for it? Regard the \( \{a, b\}^* a \) part to be \( O(1) \) length.
   How long is \( \{a, b\}^n \)?
Recall that lang \( \{ a, b \}^* a \{ a, b \}^n \).

1. DFA requires \( 2^{n+1} \) states.
2. NFA can be done with \( n + 2 \) states.
3. How long is the regex for it? Regard the \( \{ a, b \}^* a \) part to be \( O(1) \) length.
   How long is \( \{ a, b \}^n \)?
   \( \{ a, b \}^n \) is not a regex.
Textbook Regular Expressions

Recall that \( \{a, b\}^*a\{a, b\}^n \).

1. DFA requires \( 2^{n+1} \) states.
2. NFA can be done with \( n + 2 \) states.
3. How long is the regex for it? Regard the \( \{a, b\}^*a \) part to be \( O(1) \) length.
   How long is \( \{a, b\}^n \)?
   \( \{a, b\}^n \) is not a regex.
   \( \{a, b\}\{a, b\} \cdots \{a, b\} \) is a regex, so length \( O(n) \).
   However one sees things like \( \{a, b\}^n \) in textbooks all the time!
Recall that $\text{lang } \{a, b\}^* a \{a, b\}^n$.

1. DFA requires $2^{n+1}$ states.
2. NFA can be done with $n + 2$ states.
3. How long is the regex for it? Regard the $\{a, b\}^* a$ part to be $O(1)$ length.
   How long is $\{a, b\}^n$?
   $\{a, b\}^n$ is not a regex.
   $\{a, b\} \{a, b\} \cdots \{a, b\}$ is a regex, so length $O(n)$.

However one sees things like $\{a, b\}^n$ in textbooks all the time!

**Def** A **textbook regex** is one that allow exponents (formal def on next page).
Textbook Regular Expressions

Recall that lang $\{a, b\}^* a \{a, b\}^n$.

1. DFA requires $2^{n+1}$ states.

2. NFA can be done with $n + 2$ states.

3. How long is the regex for it? Regard the $\{a, b\}^* a$ part to be $O(1)$ length.
   How long is $\{a, b\}^n$?
   $\{a, b\}^n$ is not a regex.
   $\{a, b\}\{a, b\} \cdots \{a, b\}$ is a regex, so length $O(n)$.

However one sees things like $\{a, b\}^n$ in textbooks all the time!

**Def** A textbook regex is one that allow exponents (formal def on next page).

$\{a, b\}^* a \{a, b\}^n$ is a textbook regular expression of length $O(\log n)$. 
All the cool kids call them trex.

Def
All the cool kids call them trex.

Def

1. e is a trex. Every $\sigma \in \Sigma$ is a trex.
All the cool kids call them **trex**.

**Def**

1. $e$ is a trex. Every $\sigma \in \Sigma$ is a trex.
2. If $\alpha$ and $\beta$ are trex then $\alpha \cup \beta$ and $\alpha \beta$ are trex.
All the cool kids call them **trex**.

**Def**

1. $e$ is a trex. Every $\sigma \in \Sigma$ is a trex.
2. If $\alpha$ and $\beta$ are trex then $\alpha \cup \beta$ and $\alpha \beta$ are trex.
3. If $\alpha$ is a trex then $\alpha^*$ is a trex.
All the cool kids call them trex.

**Def**

1. \( e \) is a trex. Every \( \sigma \in \Sigma \) is a trex.
2. If \( \alpha \) and \( \beta \) are trex then \( \alpha \cup \beta \) and \( \alpha \beta \) are trex.
3. If \( \alpha \) is a trex then \( \alpha^* \) is a trex.
4. (This is the new step.) If \( \alpha \) is a trex and \( n \in \mathbb{N} \) then \( \alpha^n \) is a trex. We write \( n \) in binary so length is \( |\alpha| + \lg n + O(1) \).
All the cool kids call them **trex**.

**Def**

1. e is a trex. Every $\sigma \in \Sigma$ is a trex.
2. If $\alpha$ and $\beta$ are trex then $\alpha \cup \beta$ and $\alpha \beta$ are trex.
3. If $\alpha$ is a trex then $\alpha^*$ is a trex.
4. (This is the new step.) If $\alpha$ is a trex and $n \in \mathbb{N}$ then $\alpha^n$ is a trex. We write $n$ in binary so length is $|\alpha| + \lg n + O(1)$.

Clearly there is a regex for $L$ iff there is a trex for $L$. 
All the cool kids call them trex.

**Def**

1. $e$ is a trex. Every $\sigma \in \Sigma$ is a trex.
2. If $\alpha$ and $\beta$ are trex then $\alpha \cup \beta$ and $\alpha \beta$ are trex.
3. If $\alpha$ is a trex then $\alpha^*$ is a trex.
4. (This is the new step.) If $\alpha$ is a trex and $n \in \mathbb{N}$ then $\alpha^n$ is a trex. We write $n$ in binary so length is $|\alpha| + \lg n + O(1)$.

Clearly there is a regex for $L$ iff there is a trex for $L$. A trex may give a much shorter expression than a regex.
Regex vs Trex For Length

\[ L_n = \Sigma^* a \Sigma^n \]
Regex vs Trex For Length

\[ L_n = \Sigma^* a\Sigma^n \]

\( L_n \) has a length \( O(n) \) regex
$L_n = \Sigma^* a\Sigma^n$

$L_n$ has a length $O(n)$ regex

$L_n$ has a length $O(\log n)$ trex
Regex vs Trex For Length

\[ L_n = \Sigma^* a\Sigma^n \]

\( L_n \) has a length \( O(n) \) regex
\( L_n \) has a length \( O(\log n) \) trex

Need a lower bound for length of regex for \( L_n \).
Can we show that every regex for \( L_n \) requires length \( f(n) \) for some \( f(n) \) where \( \log n \ll f(n) \)?
$L_n = \Sigma^* a\Sigma^n$

$L_n$ has a length $O(n)$ regex

$L_n$ has a length $O(\log n)$ trex

Need a lower bound for length of regex for $L_n$.
Can we show that every regex for $L_n$ requires length $f(n)$ for some $f(n)$ where $\log n \ll f(n)$?

Breakout Rooms!
Assume there is a regex for $L_n$ of size $f(n)$ (we pick $f(n)$ later).
Assume there is a regex for $L_n$ of size $f(n)$ (we pick $f(n)$ later). Then there is an NFA for $L_n$ of size $f(n)$. Any DFA for $L_n$ has $\geq 2^n + 1$. Need $2f(n) < 2^n + 1$ to get a contradiction. $f(n) = n$ will suffice.

Upshot: There is a lang $L_n$ with a trex of size $O(\log n)$ but the regex requires $\geq n$. Great! We have a large size difference.
Assume there is a regex for $L_n$ of size $f(n)$ (we pick $f(n)$ later). Then there is an NFA for $L_n$ of size $f(n)$. Then there is a DFA for $L_n$ of size $2^{f(n)}$. 

Any DFA for $L_n$ has $\geq 2^{n} + 1$. Need $2^{f(n)} < 2^{n} + 1$ to get a contradiction. $f(n) = n$ will suffice. Upshot

There is a lang $L_n$ with a trex of size $O(\log n)$ but the regex requires $\geq n$. Great! We have a large size difference.
Assume there is a regex for $L_n$ of size $f(n)$ (we pick $f(n)$ later). Then there is an NFA for $L_n$ of size $f(n)$. Then there is a DFA for $L_n$ of size $2^{f(n)}$. Any DFA for $L_n$ has $\geq 2^{n+1}$. 
Assume there is a regex for $L_n$ of size $f(n)$ (we pick $f(n)$ later). Then there is an NFA for $L_n$ of size $f(n)$. Then there is a DFA for $L_n$ of size $2^{f(n)}$. Any DFA for $L_n$ has $\geq 2^{n+1}$.

Need $2^{f(n)} < 2^{n+1}$ to get a contradiction.
Assume there is a regex for \( L_n \) of size \( f(n) \) (we pick \( f(n) \) later). Then there is an NFA for \( L_n \) of size \( f(n) \). Then there is a DFA for \( L_n \) of size \( 2^{f(n)} \). Any DFA for \( L_n \) has \( \geq 2^{n+1} \). Need \( 2^{f(n)} < 2^{n+1} \) to get a contradiction. \( f(n) = n \) will suffice.
Regex vs Trex For Length: Breakout Rooms!

Assume there is a regex for $L_n$ of size $f(n)$ (we pick $f(n)$ later). Then there is an NFA for $L_n$ of size $f(n)$. Then there is a DFA for $L_n$ of size $2^{f(n)}$. Any DFA for $L_n$ has $\geq 2^{n+1}$. Need $2^{f(n)} < 2^{n+1}$ to get a contradiction. $f(n) = n$ will suffice.

**Upshot** There is a lang $L_n$ with a trex of size $O(\log n)$ but the regex requires $\geq n$. Great! We have a large size difference.
Perl Regex and Java Regex

Regex and trex:

1. PRO  Clean mathematical theory, closed under many operations

2. CON  There are many patterns we cannot express such as $L = \{ a^n b^n : n \in \mathbb{N} \}$

Perl Regex and Java Regex (which I won't define)

1. PRO  Can express many non-regular patterns such as $L$ above.

2. CON  The mathematical theory is not as clean. Maybe only people like me care.
Perl Regex and Java Regex

Regex and trex:

1. **PRO** Clean mathematical theory, closed under many operations
Regex and trex:

1. **PRO** Clean mathematical theory, closed under many operations
2. **CON** There are many patterns we cannot express such as

   \[ L = \{ a^n b^n : n \in \mathbb{N} \} \]

Perl Regex and Java Regex (which I won’t define)
Perl Regex and Java Regex

Regex and trex:

1. **PRO** Clean mathematical theory, closed under many operations
2. **CON** There are many patterns we cannot express such as $L = \{a^n b^n : n \in \mathbb{N}\}$

Perl Regex and Java Regex (which I won’t define)

1. **PRO** Can express many non-regular patterns such as $L$ above.
Perl Regex and Java Regex

Regex and trex:
1. **PRO** Clean mathematical theory, closed under many operations
2. **CON** There are many patterns we cannot express such as

\[ L = \{a^n b^n : n \in \mathbb{N}\} \]

Perl Regex and Java Regex (which I won’t define)
1. **PRO** Can express many non-regular patterns such as \( L \) above.
2. **CON** The mathematical theory is not as clean.
Regex and trex:

1. **PRO** Clean mathematical theory, closed under many operations
2. **CON** There are many patterns we cannot express such as

\[ L = \{ a^n b^n : n \in \mathbb{N} \} \]

Perl Regex and Java Regex (which I won’t define)

1. **PRO** Can express many non-regular patterns such as \( L \) above.
2. **CON** The mathematical theory is not as clean. Maybe only people like me care.
Useful!

The following algorithm is actually used in grep and other pattern recognizers.
Useful!

The following algorithm is actually used in grep and other pattern recognizers

1. Input a regex $\alpha$ which is the pattern you want to search for.
Useful!

The following algorithm is actually used in grep and other pattern recognizers:

1. Input a regex $\alpha$ which is the pattern you want to search for.
2. Create an NFA $N$ for $\alpha$ as in the last slide.
The following algorithm is actually used in grep and other pattern recognizers

1. Input a regex $\alpha$ which is the pattern you want to search for.
2. Create an NFA $N$ for $\alpha$ as in the last slide.
3. Convert the NFA $N$ to a DFA $M$ (usually the state blowup will be reasonable).
The following algorithm is actually used in grep and other pattern recognizers

1. Input a regex $\alpha$ which is the pattern you want to search for.
2. Create an NFA $N$ for $\alpha$ as in the last slide.
3. Convert the NFA $N$ to a DFA $M$ (usually the state blowup will be reasonable).
4. Run the DFA $M$ on a text to find where the pattern occurs.