Remember that a language $L$ is *regular* if there is a DFA $M$ recognizing it: $L(M) = L$, where $L(M) = \{ x \in \Sigma^* : \delta(q_0, x) \in F \}$.

We use the abbreviation REG to denote the class of all regular languages.

Another language class: \textbf{FINITE}, the class of all finite languages.

Note: \textbf{FINITE} $\subseteq$ REG.
Nondeterminism

_Nondeterminism_ is a great idea due to Michael Rabin. One way to think about it is allowing an automaton to “choose” between a number of competing transitions.

For example, suppose we had \( \delta(q_0, a) = \{r, s\} \). Then if the automaton is in state \( q_0 \), and reads the symbol \( a \), it could “choose” between going to state \( r \) and state \( s \).

This introduces a problem. What if \( r \) is an accepting state and \( s \) is not? Is the input string \( a \) accepted or not?

We don’t want to have inconsistent results depending on how the automaton chooses, so we need a rule: **In a nondeterministic machine, an input is accepted if there exists some path from the initial state to a final state.**

There could be other paths that lead from the initial state to nonaccepting states. That doesn’t matter.

So an input is _not_ accepted if _no_ path leads from the initial state to a final state.
Nondeterminism

A way to depict the situation in the previous slide is with a transition diagram, as follows:

Here, from state $q_0$, on input $a$ we have a choice: to go to either state $r$ or state $s$. Then the input $a$ is accepted, because some choice (namely, the choice to go to state $r$) leads to an accepting state.

A finite automaton that uses nondeterminism is called a *nondeterministic finite automaton* or just NFA.
Why is nondeterminism useful?

It doesn’t change the class of languages accepted: as we’ll see, both nondeterministic finite automata and deterministic finite automata accept the same class of languages.

But nondeterminism lets you recognize some languages more “naturally”, and sometimes using fewer states.

Example: binary strings that have 111 as a substring:
Another example: binary strings with a 1 occurring 5 positions from the end:

Here an NFA can recognize this language using 6 states, but no DFA can recognize the same language using less than 32 states.

In general the language

$$L = (0 \cup 1)^*1(0 \cup 1)^{n-1}$$

can be recognized by an NFA with \(n + 1\) states, but any DFA recognizing it needs at least \(2^n\) states.

So NFA’s can be *exponentially more concise* in their description of some regular language.
(Note: our definition of NFA is slightly different from John Watrous’ notes; he allows $\epsilon$-transitions. I think it’s clearer to start with a more basic NFA model, and then modify it later to allow $\epsilon$-transitions.)

To define an NFA, two things need to change from the corresponding definitions for DFA:

- the range of the transition function $\delta$, and
- the formal definition of acceptance

For an NFA the range of $\delta$ is $2^Q$, the set of all subsets of $Q$. (Also written as $\mathcal{P}(Q)$.)
Example of definition of $\delta$

<table>
<thead>
<tr>
<th>state</th>
<th>input</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>$q_0$</td>
<td>$q_0, q_1$</td>
<td></td>
</tr>
<tr>
<td>$q_1$</td>
<td>$q_2$</td>
<td>$q_2$</td>
<td></td>
</tr>
<tr>
<td>$q_2$</td>
<td>$q_3$</td>
<td>$q_3$</td>
<td></td>
</tr>
<tr>
<td>$q_3$</td>
<td>$q_4$</td>
<td>$q_4$</td>
<td></td>
</tr>
<tr>
<td>$q_4$</td>
<td>$q_5$</td>
<td>$q_5$</td>
<td></td>
</tr>
<tr>
<td>$q_5$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td></td>
</tr>
</tbody>
</table>

The diagram shows a 5-state automaton with transitions labeled 0, 1, and 0.1.
We need to define an extended transition function $\delta^*$.

The intent is that $\delta^*(q, x)$ is the set of all possible states the NFA could be in, when starting in state $q$ and reading $x$.

How shall we define $\delta^*$? Again, use a recursive definition.

The simplest case is $|x| = 0$, that is, $x = \epsilon$. In this case the most natural definition is

$$\delta^*(q, \epsilon) = \{q\}.$$

Now we need to define $\delta^*(q, xa)$, where $x$ is a string and $a$ is a single symbol.

Pause the video now and try to write the definition on your own.
Extended transition function for NFA

$$\delta^*(q, xa) = \bigcup_{r \in \delta^*(q, x)} \delta(r, a)$$
Finally, we have to define acceptance.

Remember that $\delta^*(q, x)$ is a set.

We want to accept if this set has at least one final state.

So we define

$$L(M) = \{ x \in \Sigma^* : \delta^*(q, x) \cap F \neq \emptyset \}.$$
Understanding NFA’s

- NFA’s are *not* probabilistic or randomized machines.
- NFA’s are not really *parallel* machines. There are some similarities, however.
- The computation of an NFA can be thought of as “guessing” an acceptance path, and then verifying that the guess was correct.
  - If an input is not accepted, then no guess can be verified.
- NFA’s are *not* meant to be realistic models of actual physical machines. Instead, they are a conceptual model that simplifies thinking about some issues.
Michael O. Rabin (1931–) is an Israeli mathematician and computer scientist. He is generally credited with inventing the idea of nondeterminism, in a fundamental paper on automata theory, written with Dana Scott, in 1959.

Together with Gary Miller, he is the inventor of the Miller-Rabin test for primality.

He and Dana Scott won the Turing award together in 1976.
Big theorem: *A language is regular iff it is recognized by an NFA.*

One direction is easy: by definition, a language is regular if it is recognized by a DFA. But a DFA is an NFA; just an NFA that doesn’t use any nondeterminism.

For the other direction, we need to show that if $L$ is recognized by an NFA, then it is recognized by some DFA.

Big idea: simulate an NFA with a DFA.

The DFA must somehow keep track of all the states that the original NFA could be in. It does this by having its states be sets of states of the original NFA.

This is called the *subset construction.*
Simulate NFA with DFA

NFA

\[ M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1) \]

simulating DFA

\[ M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2) \]

\[ Q_2 = 2^{Q_1} \]

\[ q_2 = \{ q_1 \} \]

\[ F_2 = \{ S \in Q_2 : S \cap F_1 \neq \emptyset \} \]

How to define \( \delta_2 \)?

We want \( \delta_1^*(q_1, x) = \delta_2^*(q_2, x) \) for all \( x \), but we can only define \( \delta_2 \) and get \( \delta_2^* \) from it. So we need to define \( \delta_2 \) to make the desired equality true.

Pause the video and try to write down a definition of \( \delta_2 \) in terms of \( \delta_1 \).
Here’s the solution:

$$\delta_2(S, a) := \bigcup_{r \in S} \delta_1(r, a).$$
Now we want to prove that our simulation works, i.e., that $L(M_1) = L(M_2)$.

This is done in the next video. But before you watch it, spend 20 minutes or so trying to construct a proof on your own.

Hint: one big step is to prove that $\delta^*_1(q_1, x) = \delta^*_2(q_2, x)$. 