A language represents:

- A logical condition on a string
- A yes/no decision problem
- With an (arbitrary but) specific encoding of objects as strings
- Of two powers of 2?

Instances:

- \( N = 7^2 \): Yes: \( 2^3 - 2^0 = 8 - 1 = 7 \)
- \( N = 120^2 \): Yes: \( 2^7 - 2^3 = 128 - 8 = 120 \)
- \( N = 26^2 \): No!
- \( N = 54^2 \): No.
- \( N = 1,260^3 \)
- \( 7 = 111 \)
- \( 120 = 1111000 \)
- \( 54 = 110110 \)
- \( 1260 = \ldots \)
- \( 8 = \ldots \)

Algorithm: The answer is Yes unless a '1' follows a '0' in \( n \), under standard binary encoding (Note: \( n > 0 \)).

I actually gave a logical assertion that implies an algorithm and states its correctness. Prove it?

"Proof by Picture":

\[
2^k = \ldots 10000 \ldots 0
\]

For some \( j < k \),

\[
2^j = \ldots 1111000 \ldots 0
\]

Output always has one or more 1's followed by zero or more 0's.

Shorthand: Binary \( (n) = \lfloor 1.1^*0^* \rfloor = 1^*0^* \) Superscript + means one or more.
Part II: A machine-type algorithm for this problem.

Language $L = 1^+ 0^*$. $0^* = 1^+ 0^* = \{x \in \{0, 1\}^* : x$ begins with at least one $1$ and has $1$s followed optionally by $0$s $\}$.

$L = \{x : x$ is a standard binary encoding of a number that is not a digit of two powers of $2$, or $x$ is not a legal encoding $\}$.

Formal Definition 6.1.1: "Nirvana state".

A deterministic finite automaton (DFA) is a 5-tuple $M = (Q, \Sigma, S, s, F)$ where:

- $Q$ is a finite set of elements called states.
- $\Sigma$ is the input alphabet.
- $S$, a member of $Q$, is the start state ($s$).
- $F$, a subset of $Q$, is the set of final states.
- $\delta : Q \times \Sigma \rightarrow Q$. Here:

$\delta(0, 0) = 4$  $\delta(0, 1) = 2$  
$\delta(1, 0) = 9$  $\delta(1, 1) = 2$

$Q = \{0, 1, 2, 3, 4\}$  $\Sigma = \{0, 1\}$  
$S = \{4\}$  $F = \{3\}$ in $M'$  $\delta(0, 1) = 5$  $\delta(1, 1) = 6$

Desired final states:

For $M$, $F = \{2, 3\}$  
$Q = \{1, 2, 3, 4\}$

For $M'$, $F' = \{1, 3\}$ instead.

Given char and a state type already known:

- set $\langle$state$\rangle$ $Q$;
- set $\langle$char$\rangle$ $\Sigma$;
- State $S$;
- set $\langle$state$\rangle$ $F$;
- State [*delta*] (State $p$),

Member function rather than a method.

State $\delta$ (State $p$, char $c$);
Segue to Thursday's lecture

Example 2: \( L_2 = \{ x \in \{0,1\}^*: \text{the number of 1s in } x \text{ is even} \} \)

Unlike Example 1, I don't know a simple way to define the corresponding set of binary numbers "naturally".

The algorithm is easy though — a DFA with only 3 states will do:

- \( Q = \{ \text{Even}, \text{Odd} \} \)
- \( \Sigma = \{0,1\} \)
- \( \delta(\text{Even},0) = \text{Even} \)
- \( \delta(\text{Even},1) = \text{Odd} \)
- \( \delta(\text{Odd},0) = \text{Odd} \)
- \( \delta(\text{Odd},1) = \text{Even} \)

Using the idea of the graph of a function, we can rewrite \( \delta \) as a set of instructions:

\[
\text{Set}(\text{State}, \text{char}, \text{State}) \text{ delta} = \{ (\text{Even},0,\text{Even}), (\text{Even},1,\text{Odd}), (\text{Odd},0,\text{Odd}), (\text{Odd},1,\text{Even}) \} \]

Note we defined the whole DFA without using numbers for states: State can be anything. To be super-technical we should write the graph form of \( \delta \) as a nested pair \(( (\text{Even}, 1), \text{Odd} ) \), but simple triples are AOK. C++/Java/... let you do it either way too:

Let's set \( \text{Set<Triple<State, char, State>>} \) or \( \text{Set<Pair<Pair<State, State>>, State>>} \)

C++ does not have a standard Triplet type. You could write your own "Set<T" class to be like above. "Yuck!" (use typedef?) in C++ need this space. (OK...)

A regular expression for \( L_2 \) is easy too. We can reason directly or by "tracing" the DFA.

- Directly: "0" doesn't matter, so we can stick "zero or more 0s" anywhere. What we need is "zero or more pairs of 1s." By itself that's \((11)^*\). Combined gives \((0*10*10)^*\)

- From the machine: We need to begin and end at the "Even" state. We can come back in two ways: a single 0; or a 1 then any number of 0s then a 1 again. Expression:

\[ \text{Comeback} = 0 \cup 10^*1. \]

We can and need to do zero-or-more "Comebacks," so we get: \((0 + 10^*1)^*\)

Hey! The answers are different! Is one wrong? No — but it's a sign of a "bumpy ride..."