EQ DFA: Instance: Two DFA $M_1$ and $M_2$ "Two Machines"

Question: Is $L(M_1) = L(M_2)$?

DFA Intersection Nonemptiness Inst: Same. Question: Is $L(M_1) \cap L(M_2) \neq \emptyset$?

Algorithm for 2: Apply the Cartesian product construction (for $\Delta$) to build a DFA $M_3$ such that $L(M_3) = L(M_1) \triangle L(M_2)$.

- Feed $M_3$ to our algorithm for $\text{NEQFA}$, and copy back the answer.

$L(M_1) \cap L(M_2) \neq \emptyset \iff L(M_3) \neq \emptyset$.

Algorithm for 1: $L(M_3) = \emptyset \iff L(M_1) = L(M_2)$.

$\Delta \Omega = A \cup B \cup \Delta \cup A$

$= A \cup B \supseteq A \cap B$

* Similar algorithm with $M_3 = \text{the Cartesian DFA for } \Delta$.*

EQ Regex: Inst: Two regexps $r_1$ and $r_2$.

Question: Is $L(r_1) = L(r_2)$?

Algorithm: Convert $r_1 \rightarrow \text{NFA}_1 \rightarrow \text{DFA}_M_1$

$r_2 \rightarrow \text{NFA}_2 \rightarrow \text{DFA}_M_2$

Then feed to $\text{EQ DFA}(M_1, M_2)$, and copy back the same answer (not the inverted answer).

Either way, all conversion steps are computable (nice but can be nasty exponential blowup).

How about EQ EFG? Inst: Two CFGs $G_1, G_2$. Question: Is $L(G_1) = L(G_2)$?

$\Delta$ ALL-REG is the special case where $L(G_2) = \Sigma^*$. 
Acceptance Problems: \( M = (Q, \Sigma, \delta, s, F) \) Type: A machine and a string.

A DFA: Instance: A DFA \( M \) and a string \( x \in \Sigma^* \)
Question: Does \( M \) accept \( x \)?
Algorithm: "Just run it." \( M(x) \), halts within \( n \) steps, and basically so will your algorithm — more exact.

A NFA: Instance: NFA \( N \), input \( x \)
Question: Does \( N \) accept \( x \)?
Algorithm: 1. Convert \( N \) to DFA \( M \), feed \( \langle M, x \rangle \) to A DFA alg.
   \( M = (Q', \Sigma, \delta', s', F') \)
   \( |Q'| \cdot |\Sigma | \cdot n \) steps, but \( |Q'| \) can be as big as \( 2^{\ell Q} \).

Algorithm 2: Simulate \( N(x) \) directly by keeping track of the current possible state set after each char \( x_i \).

Runtime: \( O(L^2(n)) \): a lot better: \( \Omega_l \)
(FK: impossibly further)

A DPDA: Instance: A DPDA \( M \), an input \( x \)
Question: Does \( M \) accept \( x \)?
Algorithm: Just run \( M(x) \).
Runtime: \( O(1Qm \cdot |x|) \).

A NPDA: Instance: An NPDAN, a string \( x \)
Question: Does \( N \) accept \( x \)?

A: Can't convert \( N \) to DPDA \( M \):
\( L(N) \) might not be a DCFL.

Presence of stack makes "Alg 2" option difficult also.

Alg 3: \( \cdot \) Convert \( N \) to an equivalent CFG \( G \) [Thm in §2.3.1]
\( \cdot \) Put \( G \) in NL Chomsky NF \( G' \) (alg in text can have exponential blow-up)
\( \equiv \{(V, Z, R, S)\} \) \( \cdot \) \( S \Rightarrow^* x \) iff \( S \Rightarrow^* x \) in exactly \( 2 |x| \cdot 1 \) steps, so we can
\( \equiv \{(V, Z, R, S)\} \) \( \cdot \) Alg 6 is decidable, exhaustively try all \( 1 \cdot |x| \cdot n \) possibilities, painful but always halts.

A TM: Instance: A deterministic TM \( M \), an input \( x \)
Question: Does \( M \) accept \( x \)?
Non-acceptance: \( N_{\text{TM}} = \sim A_{\text{TM}} \) (modulo instances where the "M" part is)

"N_{\text{TM}}": Instance: a DTM M, an input \( x \rightarrow M \) not a valid TM.

Question: Does \( M \) not accept \( x \)?

Types: An M and an X

Note: We can represent a Turing machine \( M \) as any pair \( \langle \text{DTM}, \text{string} \rangle \) of:

- \( M \) (thinking of its source code)
- \( \langle M \rangle \) some unspecified encoding - text or object code etc.
- \( e(M) \) giving a name to some encoding function \( e(-) \)

historical: \( \langle \text{Me} \rangle \) or just \( e \), calling \( e \) a Gödel Number for \( M \).

Diagonal Problem \( D_{\text{TM}} \): Instance: A DTM M. Type: "Just a Machine."

Question: Does \( M \) not accept \( \langle M \rangle \)?

As a language: \( L_{\text{D}} = D_{\text{TM}} \equiv \{ e(M) : M \text{ does not accept } e(M) \} \).

For "Quixotic?"

Theorem: There does not exist a TM Q such that \( L(Q) = D_{\text{TM}} \).

Proof: Suppose Q existed. Then it would have a code \( e(Q) \).

What could Q do on input \( e(Q) \)? The logical requirements would be:

\[ Q \text{ accepts } e(Q) \iff e(Q) \in D_{\text{TM}} \iff \sim (Q \text{ does not accept } e(Q)) \]

These requirements are self-contradictory: A logical statement \( S \) can never be \( S \) and its negation \( \sim S \).

- Rollback: Q does not exist.
- The D_{\text{TM}} language is not Turing-recognizable,
- so ipso-facto it is undecidable.

If we had a decision algorithm for \( A_{\text{TM}} \), then we could do D_{\text{TM}} by:
- Feed \( \langle M, e(M) \rangle \) to A.
- Loop back the opposite answer.
There does exist a de Bruijn machine $U$ such that

$$L(U) = \text{Arm} = \frac{2}{3} \langle M, x \rangle : M \text{ is a DTM that accepts } x.$$ 

**Proof:**

- Take the "Turing Kit" Java program which lets you enter $M$ and $x$ and simulate $M(x)$. 
  - "String accepted." 
  - Compile Turing Kit into a mini assembly program $P$. 
  - Hard-code $P$ as input to my "Universal RAM simulator". 
  - New type has $P M \# x$ enter simulator.

This can be $U$.

Arm is Turing-recognizable. 

Why doesn't $U$ make Arm decidable? 

$U(M, x)$ won't halt if $M(x)$ doesn't halt. Hence it might never give a (yes or no) answer that you can copy back as "yes".

**Diagram:**

- Arm (AR) 
- RE (Computable) 
- Co-RE (Co-computable) 
- REC (Recursive) 
- Decidable 
- Co-RE

**Definition:** $\text{RE} = \exists \text{ languages } L:$

- $L$ is computable enumerable?
- $\text{Co-RE} = \exists \not L: L \in \text{RE}$?
- $\text{REC} = \{ L: L \text{ is decidable } \}$
- $\text{Dec} = \{ L: L \text{ is decidable } \}$

Thus: $= \text{RE} \cap \text{Co-RE}$