UMB CS420

Turing Machines (TMs)

Monday, April 1, 2024
Announcements

noon 12pm
out HW

12pm/noon due Mon
in HW
Announcements

• HW 6 in
  • due Mon 4/1 12pm noon

• HW 7 out
  • due Mon 4/8 12pm noon
CS 420: Where We’ve Been, Where We’re Going

- **PDAs:** recognize context-free languages
  - Memory: states + infinite stack (push/pop only)
  - Can’t express: arbitrary dependency,
    - e.g., \( \{ww | w \in \{0,1\}^*\} \)
- **DFAs / NFAs:** recognize regular langs
  - Memory: finite states
  - Can’t express: dependency
    e.g., \( \{0^n1^n | n \geq 0\} \)
CS 420: Where We’ve Been, Where We’re Going

• **Turing Machines (TMs)**
  - **Memory**: states + infinite tape, (arbitrary read/write)
  - Expresses any “computation”

• **PDAs**: recognize context-free languages
  - **Memory**: states + infinite stack (push/pop only)
  - Can’t express: arbitrary dependency,
    - e.g., \( \{ww \mid w \in \{0,1\}^*\} \)

• **DFAs / NFAs**: recognize regular langs
  - **Memory**: finite states
  - Can’t express: dependency
    - e.g., \( \{0^n1^n \mid n \geq 0\} \)

Turing-recognizable
decidable
context-free

A special subset of TMs

regular
DCFLs
Alan Turing

• First to formalize a model of computation
  • I.e., he invented many of the ideas in this course

• And worked as a codebreaker during WW2

• Also studied Artificial Intelligence
  • The Turing Test

ChatGPT passes the Turing test

In 1950, Alan Turing proposed the Turing Test as a way to measure a machine’s intelligence. The test pits a human against a machine in a conversation. If the machine can fool the human into thinking it is also human, then it is said to have passed the test. In December 2022, ChatGPT, an artificial intelligence chatbot, became the second chatbot to pass the Turing Test, according to Max Woolf, a data scientist at BuzzFeed.

Google’s LaMDA AI passed the Turing test in the summer of 2022, demonstrating that it is invalid. For many years, the Turing test has been used as a standard for sophisticated artificial intelligence models.

Congrats to OpenAI on winning the Turing Test.
Finite Automata vs Turing Machines

- **Turing Machines** can read and write to arbitrary “tape” cells
  - Tape initially contains input string

- Tape is infinite
  - To the right

- Each step: “head” can move left or right

- Turing Machine can accept / reject at any time

Call a language *Turing-recognizable* if some Turing machine recognizes it.
Turing Machine Example

Define: $M_1$ accepts inputs in language $B = \{w\#w \mid w \in \{0,1\}^*\}$

$M_1 = \text{"On input string } w:\text{"

1. Zig-zag across the tape to corresponding positions on either side of the \# symbol to check whether these positions contain the same symbol. If they do not, or if no \# is found, reject. Cross off symbols as they are checked to keep track of which symbols correspond.

High-level: “Cross off”
Low-level $\delta$: write “x” char

This is a high-level TM description

It is equivalent to (but more concise than) our typical (low-level) tuple descriptions, i.e., one step = maybe multiple $\delta$ transitions

Analogy
“High-level”: Python
“Low-level”: assembly language
Turing Machine Example

$M_1$ accepts inputs in language $B = \{w\#w \mid w \in \{0,1\}^*\}$

$M_1 = \text{"On input string } w:\n1. \text{ Zig-zag across the tape to corresponding positions on either side of the } \# \text{ symbol to check whether these positions contain the same symbol. If they do not, or if no } \# \text{ is found, reject.}\n\text{Cross off symbols as they are checked to keep track of which symbols correspond.}\n$
Turing Machine Example

$M_1$ accepts inputs in language $B = \{ w\#w \mid w \in \{0,1\}^* \}$

$M_1 = \text{"On input string } w:\text{"

1. Zig-zag across the tape to corresponding positions on either side of the # symbol to check whether these positions contain the same symbol. If they do not, or if no # is found, reject. Cross off symbols as they are checked to keep track of which symbols correspond.
Turing Machine Example

$M_1$ accepts inputs in language $B = \{w\#w \mid w \in \{0,1\}^*\}$

$M_1 = \text{“On input string } w:\text{“}$

1. Zig-zag across the tape to corresponding positions on either side of the # symbol to check whether these positions contain the same symbol. If they do not, or if no # is found, reject. Cross off symbols as they are checked to keep track of which symbols correspond.

Head “zags” back to start
Turing Machine Example

$M_1$ accepts inputs in language $B = \{ w\#w \mid w \in \{0,1\}^* \}$

$M_1 =$ "On input string $w$:
1. Zig-zag across the tape to corresponding positions on either side of the # symbol to check whether these positions contain the same symbol. If they do not, or if no # is found, reject. Cross off symbols as they are checked to keep track of which symbols correspond.

Continue crossing off
Turing Machine Example

$M_1$ accepts inputs in language $B = \{ w\#w \mid w \in \{0,1\}^* \}$

$M_1 =$ “On input string $w$:

1. Zig-zag across the tape to corresponding positions on either side of the # symbol to check whether these positions contain the same symbol. If they do not, or if no # is found, *reject*. Cross off symbols as they are checked to keep track of which symbols correspond.

2. When all symbols to the left of the # have been crossed off, check for any remaining symbols to the right of the#. If any symbols remain, *reject*; otherwise, *accept.*”
Turing Machine Example

$M_1$ accepts inputs in language $B = \{ w#w \mid w \in \{0,1\}^* \}$

$M_1 = \text{“On input string } w:\n\begin{enumerate}
\item Zig-zag across the tape to corresponding positions on either side of the # symbol to check whether these positions contain the same symbol. If they do not, or if no # is found, reject. Cross off symbols as they are checked to keep track of which symbols correspond. 
\item When all symbols to the left of the # have been crossed off, check for any remaining symbols to the right of the #. If any symbols remain, reject; otherwise, accept.”
\end{enumerate}$
A *Turing machine* is a 7-tuple, \((Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})\), where \(Q, \Sigma, \Gamma\) are all finite sets and

1. \(Q\) is the set of states,
2. \(\Sigma\) is the input alphabet not containing the *blank symbol* \(\sqcup\),
3. \(\Gamma\) is the tape alphabet, where \(\sqcup \in \Gamma\) and \(\Sigma \subseteq \Gamma\),
4. \(\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}\) is the transition function,
5. \(q_0 \in Q\) is the start state,
6. \(q_{\text{accept}} \in Q\) is the accept state, and
7. \(q_{\text{reject}} \in Q\) is the reject state, where \(q_{\text{reject}} \neq q_{\text{accept}}\).
A Turing machine is a 7-tuple, \((Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})\), where

1. \(Q\) is the set of states,
2. \(\Sigma\) is the input alphabet not containing the blank symbol \(\_\),
3. \(\Gamma\) is the tape alphabet, where \(\_ \in \Gamma\) and \(\Sigma \subseteq \Gamma\),
4. \(\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}\) is the transition function,
5. \(q_0 \in Q\) is the start state,
6. \(q_{accept} \in Q\) is the accept state, and
7. \(q_{reject} \in Q\) is the reject state, where \(q_{reject} \neq q_{accept}\).
Formal Turing Machine Example

\[ B = \{w\#w \mid w \in \{0,1\}^*\} \]

Read char (0 or 1), cross it off, move head R(right)

Transitions on this side: Crossed off a 0

A Turing machine is a 7-tuple, \((Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})\), where \(Q, \Sigma, \Gamma\) are all finite sets and

1. \(Q\) is the set of states,
2. \(\Sigma\) is the input alphabet not containing the blank symbol \(\sqcup\),
3. \(\Gamma\) is the tape alphabet, where \(\sqcup \in \Gamma\) and \(\Sigma \subseteq \Gamma\),
4. \(\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}\) is the transition function,
5. \(q_0 \in Q\) is the start state,
6. \(q_{\text{accept}} \in Q\) is the accept state, and
7. \(q_{\text{reject}} \in Q\) is the reject state, where \(q_{\text{reject}} \neq q_{\text{accept}}\).
A Turing machine is a 7-tuple, \((Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})\), where
\(Q, \Sigma, \Gamma\) are all finite sets and

1. \(Q\) is the set of states,
2. \(\Sigma\) is the input alphabet not containing the blank symbol \(\triangleright\),
3. \(\Gamma\) is the tape alphabet, where \(\triangleright \in \Gamma\) and \(\Sigma \subseteq \Gamma\),
4. \(\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}\) is the transition function,
5. \(q_0 \in Q\) is the initial state,
6. \(q_{\text{accept}} \in Q\) is the accept state, and
7. \(q_{\text{reject}} \in Q\) is the reject state, where \(q_{\text{reject}} \neq q_{\text{accept}}\).

\(B = \{w\#w \mid w \in \{0, 1\}^*\}\)
Formal Turing Machine Example

A Turing machine is a 7-tuple, \((Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})\), where

1. \(Q\) is the set of states,
2. \(\Sigma\) is the input alphabet not containing the blank symbol \(\_\_\) or \(\_\),
3. \(\Gamma\) is the tape alphabet, where \(\_\_ \in \Gamma\) and \(\Sigma \subseteq \Gamma\),
4. \(\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}\) is the transition function,
5. \(q_0 \in Q\) is the start state,
6. \(q_{\text{accept}} \in Q\) is the accept state, and
7. \(q_{\text{reject}} \in Q\) is the reject state, where \(q_{\text{reject}} \neq q_{\text{accept}}\).

\[
B = \{w\#w \mid w \in \{0,1\}^*\}
\]
A *Turing machine* is a 7-tuple, \((Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})\), where

1. \(Q\) is the set of states,
2. \(\Sigma\) is the input alphabet not containing the *blank symbol* \(\_\),
3. \(\Gamma\) is the tape alphabet, where \(\_ \in \Gamma\) and \(\Sigma \subseteq \Gamma\),
4. \(\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}\) is the transition function,
5. \(q_0 \in Q\) is the start state,
6. \(q_{\text{accept}} \in Q\) is the accept state, and
7. \(q_{\text{reject}} \in Q\) is the reject state, where \(q_{\text{reject}} \neq q_{\text{accept}}\).

\[B = \{w\#w \mid w \in \{0, 1\}^*\}\]

Formal Turing Machine Example

Read char (0 or 1), cross it off, move head R(right)
A Turing machine is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where $Q$, $\Sigma$, $\Gamma$ are all finite sets and

1. $Q$ is the set of states,
2. $\Sigma$ is the input alphabet not containing the blank symbol $\_\_\_\_\_\_\_\_\_\_$,
3. $\Gamma$ is the tape alphabet, where $\_ \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $q_{\text{accept}} \in Q$ is the accept state, and
7. $q_{\text{reject}} \in Q$ is the rejects state, where $q_{\text{reject}} \neq q_{\text{accept}}$.

$$B = \{w\#w \mid w \in \{0, 1\}^*\}$$

Read char ($0$ or $1$), cross it off, move head R(right)

This side: Crossed off a 1
Formal Turing Machine Example

A Turing machine is a 7-tuple, \((Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})\), where \(Q, \Sigma, \Gamma\) are all finite sets and

1. \(Q\) is the set of states,
2. \(\Sigma\) is the input alphabet not containing the blank symbol \(\_\),
3. \(\Gamma\) is the tape alphabet, where \(\_ \in \Gamma\) and \(\Sigma \subseteq \Gamma\),
4. \(\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}\) is the transition function,
5. \(q_0 \in Q\) is the start state,
6. \(q_{\text{accept}} \in Q\) is the accept state, and
7. \(q_{\text{reject}} \in Q\) is the reject state, where \(q_{\text{reject}} \neq q_{\text{accept}}\).

\[ B = \{ w\#w \mid w \in \{0,1\}^* \} \]

(transitions to) Reject state not shown (assume no write, and head moves right)
$M_1 = \text{“On input string } w:\n\begin{enumerate}
\item Zig-zag across the tape side of the # symbol to the same symbol. If the Cross off symbols as the symbols correspond.\n\item When all symbols to the left of the # have been crossed off, check for any remaining symbols remain, reject; otherwise, accept.”
\end{enumerate}$
Turing Machine: High-level Description

- $M_1$ accepts if input is in language $B = \{w\#w \mid w \in \{0,1\}^*\}$

$M_1 = \text{“On input string } w:\text{“}

1. Zig-zag across the tape to corresponding positions on either side of the # symbol to check whether these positions contain the same symbol. If they do not, or if no # is found, reject. Cross off symbols as they are examined and keep track of which symbols correspond.

2. When all symbols to the left and right of the # have been crossed off, check for any remaining symbols on either side of the #. If any symbols remain, reject; otherwise, accept.”

We will (mostly) stick to high-level descriptions of Turing machines, like this one.
TM High-level Description Tips

Analogy:
- **High-level** TM description ~ function definition in “high level” language, e.g. Python
- **Low-level** TM tuple ~ function definition in bytecode or assembly

TM high-level descriptions are **not** a “do whatever” card, some rules:
1. TM and input strings must be **named** (like function definitions)
2. Steps must be numbered
3. TM can “call” or “simulate” other TMs (if they pass appropriate arguments!)
   - e.g., step for a TM \( M \) can say: “call TM \( M_2 \) with argument string \( w \), if \( M_2 \) accepts \( w \) then ..., else ...”
   - Can split input into substrings and pass to different TMs
4. Follow typical programming “scoping” rules
   - can assume functions we’ve already defined are in “global” scope, RE2NFA ...
5. Other variables must also be defined before use
   - e.g., can define a TM inside another TM
6. **must be equivalent** to a low-level formal tuple
   - high-level “step” represents a finite # of low-level \( \delta \) transitions
   - So one step cannot run forever
   - E.g., can’t say “try all numbers” as a “step”
Non-halting Turing Machines (TMs)

• A Turing Machine can **run forever**
  • E.g., head can move back and forth in a loop

• We will work with **two classes of Turing Machines:**
  • A **recognizer** is a Turing Machine that may run forever (all possible TMs)
  • A **decider** is a Turing Machine that always halts.

---

Call a language **Turing-recognizable** if some Turing machine recognizes it.

(3 possible computation results)

Call a language **Turing-decidable** or simply **decidable** if some Turing machine decides it.

(2 possible computation results)

So: TM computation has **3 possible results:**
- Accept
- Reject
- Loop forever
Formal Definition of an “Algorithm”

- An **algorithm** is equivalent to a **Turing-decidable** Language (always halts)

Many functions we have defined this semester are algorithms! e.g., all our conversion functions are deciders!
- convertD2N
- RegExpr2NFA
- convertD2P
Turing Machine Variations
1. Multi-tape TMs

2. Non-deterministic TMs

We will prove that these TM variations are equivalent to deterministic, single-tape machines.
Reminder: Equivalence of Machines

- Two machines are **equivalent** when ... 

- ... they recognize the same language
Theorem: Single-tape TM $\Leftrightarrow$ Multi-tape TM

⇒ If a single-tape TM recognizes a language, then a multi-tape TM recognizes the language
  • Single-tape TM is equivalent to ...
  • ... multi-tape TM that only uses one of its tapes
  • (could you write out the formal conversion?)

⇐ If a multi-tape TM recognizes a language, then a single-tape TM recognizes the language
  • Convert: multi-tape TM $\rightarrow$ single-tape TM
Multi-tape TM $\Rightarrow$ Single-tape TM

**Idea:** Use delimiter (#) on single-tape to simulate multiple tapes
• Add “dotted” version of every char to simulate multiple heads
Theorem: Single-tape TM $\iff$ Multi-tape TM

⇒ If a single-tape TM recognizes a language, then a multi-tape TM recognizes the language
  • Single-tape TM is equivalent to ...
  • ... multi-tape TM that only uses one of its tapes

⇐ If a multi-tape TM recognizes a language, then a single-tape TM recognizes the language
  • Convert: multi-tape TM $\rightarrow$ single-tape TM
Nondeterministic TMs
Flashback: DFAs vs NFAs

A finite automaton is a 5-tuple \((Q, \Sigma, \delta, q_0, F)\), where
1. \(Q\) is a finite set called the states,
2. \(\Sigma\) is a finite set called the alphabet,
3. \(\delta: Q \times \Sigma \rightarrow Q\) is the transition function,
4. \(q_0 \in Q\) is the start state, and
5. \(F \subseteq Q\) is the set of accept states.

A nondeterministic finite automaton is a 5-tuple \((Q, \Sigma, \delta, q_0, F)\), where
1. \(Q\) is a finite set of states,
2. \(\Sigma\) is a finite alphabet,
3. \(\delta: Q \times \Sigma_e \rightarrow \mathcal{P}(Q)\) is the transition function,
4. \(q_0 \in Q\) is the start state, and
5. \(F \subseteq Q\) is the set of accept states.

Nondeterministic transition produces set of possible next states.
A *Turing machine* is a 7-tuple, \((Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})\), where 

- \(Q\), \(\Sigma\), \(\Gamma\) are all finite sets and
- \(Q\) is the set of states,
- \(\Sigma\) is the input alphabet not containing the *blank symbol* \(\_\),
- \(\Gamma\) is the tape alphabet, where \(\_ \in \Gamma\) and \(\Sigma \subseteq \Gamma\),
- \(\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}\) is the transition function,
- \(q_0 \in Q\) is the start state,
- \(q_{\text{accept}} \in Q\) is the accept state, and
- \(q_{\text{reject}} \in Q\) is the reject state, where \(q_{\text{reject}} \neq q_{\text{accept}}\).
Nondeterministic Turing Machine Formal Definition

A nondeterministic Turing Machine is a 7-tuple, \((Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})\), where

1. \(Q\) is the set of states,
2. \(\Sigma\) is the input alphabet not containing the blank symbol \(\_\),
3. \(\Gamma\) is the tape alphabet, where \(\_ \in \Gamma\) and \(\Sigma \subseteq \Gamma\),
4. \(\delta : Q \times \Gamma \rightarrow P(Q \times \Gamma \times \{L, R\})\)
5. \(q_0 \in Q\) is the start state,
6. \(q_{\text{accept}} \in Q\) is the accept state, and
7. \(q_{\text{reject}} \in Q\) is the reject state, where \(q_{\text{reject}} \neq q_{\text{accept}}\).
**Thm:** Deterministic TM $\iff$ Non-det. TM

$\Rightarrow$ If a **deterministic TM** recognizes a language, then a **non-deterministic TM** recognizes the language

- **Convert:** Deterministic TM $\rightarrow$ Non-deterministic TM ...
- ... change Deterministic TM $\delta$ fn output to a one-element set
  - $\delta_{ntm}(q, a) = \{\delta_{dtm}(q, a)\}$
  - (just like conversion of DFA to NFA --- HW 3, Problem 1)
- **DONE!**

$\Leftarrow$ If a **non-deterministic TM** recognizes a language, then a **deterministic TM** recognizes the language

- **Convert:** Non-deterministic TM $\rightarrow$ Deterministic TM ...
- ... ???
**Review: Nondeterminism**

Deterministic computation:
- start
- ...
- accept or reject

Nondeterministic computation:
- ...
- reject
- ...

In nondeterministic computation, every step can branch into a set of “states”

What is a “state” for a TM?

\[ \delta : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\}) \]
Flashback: PDA Configurations (IDs)

- A configuration (or ID) is a “snapshot” of a PDA’s computation.

- 3 components \((q, w, \gamma)\):
  - \(q\) = the current state
  - \(w\) = the remaining input string
  - \(\gamma\) = the stack contents

A sequence of configurations represents a PDA computation.
A Turing machine is a 7-tuple, \((Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})\), where \(Q, \Sigma, \Gamma\) are all finite sets and

1. \(Q\) is the set of states,
2. \(\Sigma\) is the input alphabet not containing the blank symbol \(\sqcup\),
3. \(\Gamma\) is the tape alphabet, where \(\sqcup \in \Gamma\) and \(\Sigma \subseteq \Gamma\),
4. \(\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}\) is the transition function,
5. \(q_0 \in Q\) is the start state,
6. \(q_{\text{accept}} \in Q\) is the accept state, and
7. \(q_{\text{reject}} \in Q\) is the reject state, where \(q_{\text{reject}} \neq q_{\text{accept}}\).
TM Configuration = State + Head + Tape
TM Configuration = State + Head + Tape

1011011111

Textual representation of “configuration” (use this in HW)

1^{st} char after state is current head position
TM Computation, Formally

\[ M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}}) \]

**Extended**
- Base Case
  \[ I \vdash^* I \text{ for any ID } I \]
- Recursive Case
  \[ I \vdash^* J \text{ if there exists some ID } K \text{ such that } I \vdash K \text{ and } K \vdash^* J \]

**Single-step**

(Right)
\[ \alpha q_1 \alpha_b \vdash \alpha \times q_2 \beta \]
if \( q_1, q_2 \in Q \)
\[ \delta(q_1, a) = (q_2, x, R) \]
\( a, x \in \Gamma \quad \alpha, \beta \in \Gamma^* \)

(Left)
\[ \alpha b q_1 \alpha_b \vdash \alpha q_2 b x \beta \]
if \( \delta(q_1, a) = (q_2, x, L) \)

**Edge cases:**
- Head stays at leftmost cell
  \[ q_1 \alpha b \vdash q_2 \times \beta \]
  if \( \delta(q_1, a) = (q_2, x, L) \)
- Add blank symbol to config
  \[ \alpha q_1 \vdash \alpha_{-} q_2 \]
  if \( \delta(q_1, \_ \_ ) = (q_2, \_ \_ R) \)

(L move, when already at leftmost cell)
(R move, when at rightmost filled cell)
Nondeterminism in TMs

Deterministic computation

- start
- ...
- accept or reject

Nondeterministic computation

For TMs, each node is a configuration

reject

accept

1011q_01111

1011q_01111
Nondeterministic TM $\rightarrow$ Deterministic

1st way

- Simulate NTM with Det. TM:
  - Det. TM keeps multiple configs on single tape
    - Like how single-tape TM simulates multi-tape
  - Then run all computations, concurrently
    - I.e., 1 step on one config, 1 step on the next, ...
  - Accept if any accepting config is found

- Important:
  - Why must we step configs concurrently?
    
    Because any one path can go on forever!
Interlude: Running TMs inside other TMs

Remember: If TMs are like function definitions, then they can be called like functions ...

Exercise:
• Given: TMs \( M_1 \) and \( M_2 \)
• Create: TM \( M \) that accepts if either \( M_1 \) or \( M_2 \) accept

Possible solution #1:
\( M = \) on input \( x , \)
1. Call \( M_1 \) with arg \( x \); accept \( x \) if \( M_1 \) accepts
2. Call \( M_2 \) with arg \( x \); accept \( x \) if \( M_2 \) accepts

Note: This solution would be ok if we knew \( M_1 \) and \( M_2 \) were deciders (which halt on all inputs)
Interlude: Running TMs inside other TMs

Exercise:
- Given: TMs $M_1$ and $M_2$
- Create: TM $M$ that accepts if either $M_1$ or $M_2$ accept

Possible solution #1:
$M = \text{on input } x,$
1. Call $M_1$ with arg $x$; accept $x$ if $M_1$ accepts
2. Call $M_2$ with arg $x$; accept $x$ if $M_2$ accepts

Possible solution #2:
$M = \text{on input } x,$
1. Call $M_1$ and $M_2$, each with $x$, concurrently, i.e.,
   a) Run $M_1$ with $x$ for 1 step; accept if $M_1$ accepts
   b) Run $M_2$ with $x$ for 1 step; accept if $M_2$ accepts
   c) Repeat

\[\begin{array}{|c|c|c|}
\hline
M_1 & M_2 & M \\
\hline
\text{reject} & \text{accept} & \text{accept} \\
\text{accept} & \text{reject} & \text{accept} \\
\text{accept} & \text{loops} & \text{accept} \\
\text{loops} & \text{accept} & \text{loops} \\
\hline
\end{array}\]
Nondeterministic TM $\rightarrow$ Deterministic

• Simulate NTM with Det. TM:
  • Number the nodes at each step
  • Check all tree paths (in breadth-first order)
    • 1
    • 1-1

\[ \text{2nd way} \quad \text{(Sipser)} \]
Nondeterministic TM $\rightarrow$ Deterministic

- Simulate NTM with Det. TM:
  - Number the nodes at each step
  - Check all tree paths (in breadth-first order)
    - 1
    - 1-1
    - 1-2
Nondeterministic TM $\rightarrow$ Deterministic

- Simulate NTM with Det. TM:
  - Number the nodes at each step
  - Check all tree paths (in breadth-first order)
    - 1
    - 1-1
    - 1-2
    - 1-1-1

2\textsuperscript{nd} way (Sipser)
Nondeterministic TM → Deterministic

Always has input, never changes

“Work tape” when checking each path (re-copy input here each time)

Tracks which node we are on, e.g., 1-1-2, etc.

Use 3 tapes

2nd way (Sipser)
Nondeterministic TM ⇔ Deterministic TM

⇒ If a deterministic TM recognizes a language, then a nondeterministic TM recognizes the language
  • Convert Deterministic TM → Non-deterministic TM

⇐ If a nondeterministic TM recognizes a language, then a deterministic TM recognizes the language
  • Convert Nondeterministic TM → Deterministic TM
Conclusion: These are All Equivalent TMs!

- Single-tape Turing Machine
- Multi-tape Turing Machine
- Non-deterministic Turing Machine