**Question:** How can we prove that for the previous problems regarding CFLs (e.g. equality, universality) there is no algorithm that solves them?

**Solution:** we need a formal definition of algorithm.

Let us start with something we know: Java.

Can we show that there is no Java program that solves these problems?

---

**Hello - World problem:**

Your first Java program: HW:

```java
public class HW {
    public static void main(String[] args) {
        System.out.println("Hello, world");
    }
}
```

The first 12 characters output by HW are "Hello, world".

**Hello - world problem (HWP):** given an arbitrary Java program P and an input I for P, does P(I) print "Hello, world" or its first 12 characters?
Consider a solution to HWP:

\[ P \quad \rightarrow \quad H \quad \rightarrow \quad \text{"yes", "no"} \]

Input \rightarrow \text{false program} \rightarrow \text{output}

Does such a program \( H \) exist?
- we could run \( P \) for print statements
- but, how do we know whether they are executed?

To give you an idea how difficult this can become, consider Fermat's last theorem:

The equation \( x^n + y^n = z^n \) has no integer solution for \( n \geq 3 \).

For \( n=2 \), a solution is \( x=3, y=4, z=5 \)

For \( n \geq 3 \), mathematicians have believed that the theorem is true, but no proof was found until recently (proof given by Wiles is very complex, and still under verification)

Consider a sample Java program \( P_1 \) that:

1) reads input \( n \)
2) for all possible \( x, y, z \) do
   
   if \( (x^n + y^n = z^n) \)
   
   println("Hello, world!")

Consider input \( n=3 \): \( P_1 \) prints "Hello, world!" only if F.L.T is false, otherwise \( P_1 \) loops forever.
If we could solve HWP, we would also have proved or disproved F.L.T.

This would be too nice!! Where is the problem?

**Theorem:** There is no finite program H that decides HWP.

**Proof:** Assume H exists and derive a contradiction.

Consider H:

```
   P  \rightarrow H
   \text{"yes"} \quad \text{\"no\"}
   I
```

We modify H to $H_1$, so that $H_4$ prints "Hello, world" instead of "no"

```
P  \rightarrow H_4
I  \rightarrow \text{"Hello, world"}
```

(Note: we have to modify the quintuple statements in H)

We modify $H_4$ to $H_2$, which takes only input $P$ and feeds it to $H_4$ as both $P$ and $I$:

```
P  \rightarrow \text{Buffer}  \rightarrow H_2
```

Java program:

```
java program

reads in P, stores it in a string
and passes it twice to $H_4$
```

Let us consider $H_2(P)$ when $P = H_2$:

- Suppose $H_2(H_2) = \text{"yes"}$ \implies $P(P) = \text{"Hello, world"}$
- Suppose $H_2(H_2) = \text{"Hello, world"}$ \implies $P(P) \neq \text{"Hello, world"}$

But $P = H_2$ \implies contradiction \implies $H, H_1, H_2$ cannot exist! Q.e.d
We have shown HWP to be undecidable, i.e., there cannot be an algorithm (or a program) that solves it.

We can show that other problems are undecidable by "reducing" HWP to them.

**Reductions**

**foo-problem:** given a program R and its input z, does R ever call a function named foo while executing on input z.

Idea: we **reduce** the HWP to the foo-problem, i.e., we show that if it's possible to solve the foo-problem on \((R, z)\), then we can solve HWP on \((Q, y)\) for any program Q with input y.

Since HWP is undecidable, so is the foo-problem.

Suppose there is a program F that takes as input \((R, z)\) and decides the foo-problem for \((R, z)\).

We show how F can be used to construct H that decides HWP on input \((R, y)\).
Idea: apply modifications to Q

1) rename function foo in Q (if present) to pipp
   $\Rightarrow Q'$
2) add a dummy function foo to $Q' \Rightarrow Q''$
3) modify $Q''$ to store all its output in some array A
   $\Rightarrow Q'''$
4) modify $Q'''$ so that after every println statement
   it checks array A to see if "Hello, world" has been
   printed. If yes, then call function foo $\Rightarrow Q''''$

Note: these modifications can be done by a Java program

Let $R = Q''''$ and $\varepsilon = \gamma$

We have by construction:

$Q(\varepsilon)$ prints "Hello, world" $\Rightarrow$

$R(\varepsilon)$ calls function foo.

Hence, we can use $F$ that solves foo-problem on $R(\varepsilon)$

to construct $H$ that solves HWFO on $Q(\varepsilon)$.

Schematically:

$$
(Q, \varepsilon) \xrightarrow{H} (R, \varepsilon) \xrightarrow{(R, \varepsilon)} F \xrightarrow{\text{"yes" or "no"}}$$

But: since $H$ does not exist, else $F$ cannot exist.
Sharing undecidability by reduction from undecidable problems

Problem \( P_2 \) taking input \( I_2 \) known to be undecidable

\[ P_1(I_1) = \text{"yes"} \iff P_2(I_2) = \text{"yes"} \]

Given solution program \( S_2 \) for \( P_2 \), we could obtain

\[ S_1 \]

Since \( S_1 \) does exist, we obtain that \( S_2 \) cannot exist

\[ \Rightarrow P_2 \text{ is undecidable.} \]

Existence of undecidable problems:

While it was tricky to show that a specific problem is undecidable, it is rather easy to show that there are infinitely many undecidable problems.

We use a counting argument:

- a problem \( P \) is a language over \( \Sigma \) (for some finite \( \Sigma \))
  (the strings in the language represent those instances of \( P \) for which the answer is "yes")
  \[ \Rightarrow \text{there are uncountably many problems} \]
- an algorithm is a string over \( \Sigma' \) (for some finite \( \Sigma' \))
  \[ \Rightarrow \text{there are countably many algorithms} \]
  \[ \Rightarrow \text{there must be (uncountably many) problems for which there is no algorithm.} \]
Some (or C, Pascal, ...) programs are not well-suited to develop a theory of computation:
- run-time environment and run-time errors
- complex language constructs
- finite memory
- "state" of the computation is complicated to represent
  would need to show that the results for a specific programming language are in fact general

=> We resort to an abstract computing device, the
  Turing Machine (TM)
  - simple and universal programming language
  - state of computation is easy to describe
  - unbounded memory
  - can simulate any known computing device

Church–Turing hypothesis:
  All reasonably powerful computation models are equivalent to TMs (but not more powerful).

=> TMs model anything we can compute.
The TM:

```
- b₁ b₂ b₃ b₄ b₅ ... e₁ e₂ e₃ ... eₙ bₘ ...
```

infinite tape

```
read/write head

Finite state control
```

Programmed by specifying transitions:

- move depends on:
  - current state (finitely many)
  - symbol under the tape head

- effects of a move:
  - new state
  - write new symbol on tape cell under the head
  - move head left/right

Observations:

- relationship to real computers: CPU = finite state control memory = tape

  "differences" (features lost in the abstraction)
  - no random access memory
  - limited instruction set

Anyway: a TM can simulate a computer (with a cubic increase in running time — see book 8.6)
Definition: $M = (Q, \Sigma, \Gamma, \delta, q_0, \emptyset, F)$

- $Q$ ... set of states (finite) $q_0 \in Q$ ... initial state
- $\Sigma$ ... input alphabet (finite) $\Gamma$ ... tape alphabet (finite)
- $F \subseteq Q$ ... final states $\emptyset \in \Gamma$ ... blank symbol

Conditions: $\Sigma \subseteq \Gamma$, since input is written initially on tape
$\emptyset \in \Gamma - \Sigma$, since the rest of the tape is blank

Initially:
- state $q_0$
- tape contains $w$ surrounded by $\emptyset$
- tape head is at the leftmost cell of the input

Transitions: $\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$

$\delta(q, \gamma) = (p, y, d)$ means that
- if $M$ is in state $q$ and tape head is over symbol $\gamma$,
  then $M$ changes state to $p$
- replaces $\gamma$ by $y$ on the tape
- moves tape head by one cell in direction $d$
  (left for $L$, right for $R$)

The TM is deterministic:
- for each $\delta(q, \gamma)$ we have at most one move
- $\delta(q, \gamma)$ could also be undefined

Acceptance: $w$ is accepted by TM $M$ if $M$, when started with $w$
on the tape, eventually enters a final state

We can assume that all final states are halting, i.e. no transition is defined for them

Rejection: $w$ rejects in non-final state (i.e., no transition defined)
- never halts (infinite loop)
Difference between FA/PDA and TM:

FA/PDA scans over w and accepts/rejects when it has reached its end.

TM can move back and forth over w and accepts/rejects when it halts on rejects if it loops forever.

Example: \[ L = \{ w^* \# w^* | w \in \{0, 1\}^+, \# \in \{0, 1, \#\}^* \} \]

Initially

TM idea: remember leftmost symbol, erase it
- move to leftmost symbol after #’s
- if the two don’t match, then reject
- otherwise replace the symbol by #, move left and start again

\[ M = (Q, \Sigma, \Gamma, \delta, q_0, \delta, \delta) \]

\[ Q = \{ q_1, q_2, \ldots, q_7 \} \]

\[ \Sigma = \{ 0, 1, \# \} \]

\[ \Gamma = \{ 0, 1, \#, \# \} \]

\[ F = \{ q_7 \} \]

\[ \delta(q_0, 0) = (q_1, \#, R) \] {erase 0 and look for matching 0

\[ \delta(q_0, 1) = (q_1, \#, R) \] { - - - - - - 1

\[ \delta(q_1, 0) = (q_1, 0, R) \]

\[ \delta(q_1, 1) = (q_1, 1, R) \]

\[ \delta(q_1, \#) = (q_3, \#, R) \] {skip over 0’s and 1’s, until # is found (remembering 0)

\[ \delta(q_2, 0) = (q_2, 0, R) \]

\[ \delta(q_2, 1) = (q_2, 1, R) \]

\[ \delta(q_2, \#) = (q_4, \#, R) \] { - - (remembering 1)

\[ \delta(q_3, 0) = (q_3, 0, L) \]

\[ \delta(q_3, 1) = (q_3, 1, L) \]

\[ \delta(q_3, \#) = (q_3, \#, L) \]
\( \delta(q_3, \#) = (q_3, \#, R) \) \{ Skip over \#'s, look for 0, 1, and replace it by \#. \}
\( \delta(q_3, 0) = (q_5, \#, L) \)

Note: if after \#'s a 0 or a 1 is found, M halts and rejects

\( \delta(q_4, \#) = (q_4, \#, R) \)
\( \delta(q_4, 1) = (q_5, \#, L) \)

As previous one, replacing 0/1 with 1/0.

\( \delta(q_5, \#) = (q_5, \#, L) \)
\( \delta(q_5, 0) = (q_6, 0, L) \)
\( \delta(q_5, 1) = (q_6, 1, L) \)
\( \delta(q_5, \Phi) = (q_7, \Phi, R) \)

More left skipping \#'s.

If to the left of the \#'s a 0 or 1 is found, move to \( q_6 \) to skip them also. If \( \Phi \) is found, accept.

\( \delta(q_6, 0) = (q_6, 0, L) \)
\( \delta(q_6, 1) = (q_6, 1, L) \)
\( \delta(q_6, \Phi) = (q_0, \Phi, R) \)

More left, skipping 0/\( \Phi \) and 1/\( \Phi \), and restart again.

Transition diagram:

```
0/\Phi \rightarrow q_0
\Phi/\Phi \rightarrow q_1
0/0 \rightarrow q_1
0/\# \rightarrow q_4
\#/\# \rightarrow q_4
0/\# \rightarrow q_3
0/\# \rightarrow q_5
\#/\# \rightarrow q_5
\Phi/\# \rightarrow q_6
0/0 \rightarrow q_0
```

Immediate description (I.D.) or configuration of a TM describes the current situation of TM and tape.

\[ I.D. = \alpha_1 \sigma \alpha_2 \quad \text{with} \quad q \in Q \]
\[ \alpha_1, \alpha_2 \in \Gamma^* \]

means:
- non-blank portion of tape contains \( \alpha_1, \alpha_2 \)
- head is on leftmost symbol of \( \alpha_2 \)
- machine is in state \( q \)

Corresponds to

\[ \text{BLANKS} \quad \alpha_1 \quad \alpha_2 \quad \text{BLANK} \]

\[ \uparrow \text{state } q \]

As for PDAs, we use \( \vdash \) and \( \vdash^* \) to denote the change of I.D. due to transitions.

Example:

\[ q_0 \alpha_1 \# \alpha_1 \vdash q_1 \alpha_1 \# \alpha_1 \vdash q_2 \alpha_1 \# \alpha_1 \vdash q_3 \alpha_1 \# \alpha_1 \vdash q_4 \alpha_1 \# \alpha_1 \vdash q_5 \alpha_1 \# \alpha_1 \vdash q_6 \alpha_1 \# \alpha_1 \vdash q_7 \alpha_1 \# \alpha_1 \vdash q_8 \alpha_1 \# \alpha_1 \vdash \]

\( \alpha \) accepts

Formal definition of language accepted by a TM \( M \):

\[ L(M) = \{ w \in \Sigma^* \mid q_0 w \vdash^* \alpha_1 \alpha_2 \quad \text{with} \quad q \in F \quad \text{and} \quad \alpha_1, \alpha_2 \in \Gamma^* \} \]
Notes:

1) We have used TMs for language recognition, which in turn corresponds to solving decision problems.
   - We can, however, consider also TMs as computing functions.
     - The output (result of the function) is left on the tape.

2) The class of languages accepted by TMs are called recursively enumerable.
   - For a string $w$ in the language
     - The TM halts on input $w$ in a final state.
   - For a string $w$ not in the language
     - The TM may halt in a non-final state, or
     - It may loop forever.

These languages for which the TM always halts (regardless of whether it accepts or not) are called recursive.
   - These languages correspond to recursive functions.
   - TMs that always halt are a good model of algorithms and they correspond to decidable problems.
We present some notational conveniences that make it easier to write TM programs.

Idea: use structured states and tape symbols

1) Storage in the state: ("CPU register")

Idea: state names are a tuple of the form

\[ [q, D_1, \ldots, D_k] \]

- \( D_i \) is the stored symbol
- \( q \) is control portion of the state

Example: TM \( M = (Q, \Sigma, \Gamma, \delta, q_0, q, F) \) for \( L = 01^*10^* \)

Idea: \( M \) remembers the first symbol and checks that it does not reappear

\[ Q = \{ [q_i, e] \mid i \in \{0,1\}, e \in \{0,1,\_\} \} = \]

\[ \{ [q_0, \_], [q_0, 0], [q_0, 1], [q_1, \_], [q_1, 0], [q_1, 1] \} \]

\[ \Sigma = \{0, 1\} \]

\[ \Gamma = \{0, 1, \_\} \]

\[ q_0 = [q_0, 0] \]

\[ q_1 = [q_1, 1] \]

\[ F = \{ [q_1, \_] \} \]

Meaning of \( [q_i, e] \)

- Control portion \( q_i \):
  - \( q_0 \): \( M \) has not yet read its first symbol
  - \( q_1 \): \( M \) has read its first symbol

- State portion \( e \): \( e \) is the first symbol read
transitions:
\[ \delta([q_0, z], e) = ([q_1, z], e, R) \] for \( e \in \{0, 1\} \)

- M remembers in \([q_0, z]\) that it has read \( e \)
\[ \delta([q_1, 0], 1) = ([q_1, 0], 1, R) \] \( M \) moves right, as long as it does not see the first symbol
\[ \delta([q_1, 1], 0) = ([q_1, 1], 0, R) \]
- \( M \) expects when it reaches the first \( b \)

2) Multiple tracks:

Idea: view tape as having multiple tracks, i.e. \( \Gamma \) in each symbol in \( \Gamma \) has multiple components

<table>
<thead>
<tr>
<th>( )</th>
<th>0</th>
<th>*</th>
<th>( b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( a )</td>
<td>( a )</td>
<td>( i )</td>
<td></td>
</tr>
</tbody>
</table>

the symbols on the tape are \([0, \_], [\_0, \_], [\_1, \_]\)

Example: \( L = \{ww \mid w \in \{0, 1\}^+\} \)

We first need to find midpoint, and then we can match corresponding symbols.

To find midpoint: we view tape as 2 tracks

\[ \begin{array}{cccc}
0 & 1 & 1 & 0 & 1 & 1
\end{array} \]

< used to put markers on symbols

Hence: \( \Gamma = \{[0, \_], [\_0, \_], [\_1, \_], [\_, \_], [\_, \_]\} \)

(note: we need no * on \( b \))
We put markers on two outermost symbols and move them inwards:

\[ \delta(q_0, [\#, R]) = (q_1, [\#, R], R) \]  \text{ move right till end on first marked symbol}
\[ \delta(q_1, [\#, L]) = (q_2, [\#, L], R) \]  \text{ move rightmost mark one symbol to the left}
\[ \delta(q_2, [\#, R]) = (q_3, [\#, R], L) \]  \text{ move left till end on first marked symbol}
\[ \delta(q_3, [\#, L]) = (q_0, [\#, L], R) \]  

Note: we have each of the above for \( i \in \{0, 1\} \)

At the end: head is over first symbol of second \( w \), with \( * \) above it, in state \( q_0 \).

3) Subroutines / procedure calls

Example: shifting over

\[ \text{given: } ID_1 = \alpha \, q_i \, \beta \]
\[ \text{want: } ID_2 = \alpha \, \# \, q_i \, \beta \]

Subroutine for shifting over can be used repeatedly to create space in the middle of the tape

E.g. to implement a counter

\[ \$0\$ \rightarrow \$1\$ \rightarrow \$\#1\$ \rightarrow \$01\$ \rightarrow \$10\$ \rightarrow \]
\[ \rightarrow \$11\$ \rightarrow \$\#11\$ \rightarrow \$011\$ \rightarrow \ldots \]
Procedure call: \[ \delta(q_i, X) = ([p, \text{X}], \begin{array}{c} \text{\text{\textbackslash r}} & \text{\text{\textbackslash l}} \end{array}, \text{R}), \forall X \in \Gamma \]

- remember return state \( q_i \), and erased symbol \( X \)
- state \( p \) calls procedure

Procedure \( p \) for shifting

1) Shift \( p \) cell to the right
   \[ \delta([p, \text{X}], \gamma) = ([p, \text{\gamma}], R), \forall \gamma, \chi \in \Gamma \text{ with } \gamma \neq \text{\textbackslash r} \]
2) till we have reached end of \( \beta \)
   \[ \delta([p, \text{\gamma}], \text{\textbackslash l}) = (\pi, \text{\gamma}, \text{L}), \forall \gamma \in \Gamma \]
3) return to calling point by moving left
   \[ \delta(\pi, \gamma) = (\pi, \gamma, \text{L}), \forall \gamma \neq [\text{\textbackslash r}] \]
4) wind and return to state \( q_i \)
   \[ \delta(\pi, [\text{\textbackslash r}]) = (q_i, \text{\textbackslash r}, \text{R}) \]

In fact, we can implement arbitrary complex procedures, with any kind of parameter passing

**Exercise:** redesign the TMs you have seen so far to take advantage of storage in the state, multiple tracks, and subroutines
Extensions to the basic TM

Note: if the TM seen so far can compute all that can be computed, then it should not become more expressive by extending it.

We consider two extensions:
- multiple tapes
- nondeterminism

and show that both can be captured by the basic T.M.

1) Multi-tape T.M.

Initially: input $w$ is on tape 1 with tape-head on the leftmost symbol. Other tapes are all blank.

Transitions: specify behaviour of each head independently

$$S(q_1, x_1, \ldots, x_h) = (q_1, (y_1, d_1), \ldots, (y_h, d_h))$$

$x_i$: symbol under head $i$
$y_i$: new symbol written to head $i$
$d_i$: direction in which head $i$ moves
To simulate a 2-tape TM $M_k$ with a 1-tape TM $M_1$,
we use $2k$ tracks in $M_1$: for each tape of $M_k$
- one track of $M_1$ to store tape content
- one track of $M_1$ to mark head position with $\ast$

<table>
<thead>
<tr>
<th>A B A C B A</th>
<th>Tape 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Head 1</td>
</tr>
<tr>
<td>0 0 1 1 1 0</td>
<td>Tape 2</td>
</tr>
<tr>
<td>*</td>
<td>Head 2</td>
</tr>
<tr>
<td>b b a b a b</td>
<td>Tape 3</td>
</tr>
<tr>
<td>*</td>
<td>Head 3</td>
</tr>
</tbody>
</table>

Each transition of $M_k$ is simulated by a series of transitions of $M_1$: $\delta(q, x_1, \ldots, x_k) = (p, (y_1, d_1), \ldots, (y_k, d_k))$
- start at leftmost head position marker
- sweep right and remember in appropriate "CPU registers" the symbols $x_i$ under each head (note: there are exactly $k$, and hence finitely many)
- knowing all $x_i$'s, sweep left, change each $x_i$ to $y_i$, and move the marker for tape $i$ according to $d_i$

Note: $M_1$ needs to remember always how many of the $k$ heads are to its left (uses an additional CPU register)

The final states of $M_1$ are those that have in the state-component a final state of $M_k$.

We can verify that we can construct $M_1$ so that $L(M_1) = L(M_k)$

(details are straightforward, but cumbersome)
Simulation Speed:

Note: Enhancements do not affect the expressive power of e TM - they do affect its efficiency.

Definition: e TM is said to have running time $T(n)$ if it halts within $T(n)$ steps on all inputs of length $n$.

Note: $T(n)$ could be infinite.

Theorem: If $M_k$ has running time $T(n)$, then $M_k$ will simulate it with running time $O(T(n)^2)$.

Proof: Consider input $w$ of length $n$.
- $M_k$ runs at most $T(n)$ time on it.
- At each step, leftmost and rightmost heads can drift apart by at most 2 additional cells.
- It follows that after $T(n)$ steps, the $k$ heads cannot be more than $2 \cdot T(n)$ apart, and $M_k$ uses $\leq 2 \cdot T(n)$ tape cells.

Consider $M_k$:
- makes two sweeps for each transition of $M_k$
- each sweep takes at most $O(T(n))$
- number of transitions of $M_k$ is $\leq T(n)$

It follows that the total running time is $O(T(n)^2)$. 

2) Non-deterministic TM (NTM)

In a (deterministic) TM, \( \delta(q, x) \) is unique or undefined.
In a NTM, \( \delta(q, x) \) is a finite set of triples,
\[
\delta(q, x) = \{(p_1, y_1, d_1), \ldots, (p_k, y_k, d_k)\}
\]

At each NP, the NTM can non-deterministically choose which transition to make.

As for other ND devices: a string \( w \) is accepted if the NTM has at least one execution leading to a final state.

**Example:** \( \Sigma = \{0, 1, \ldots, 3\} \)

\[ L = \{w \in \Sigma^* | \exists 1 \text{ appears in positions to the left of some } i, n \text{, with } 0 < i \leq 3 \} = \]

\[ \{w \in \Sigma^* | \exists j > 0 \text{ s.t. } w_{j-1} = 0 \} \]

(\( w_i \) indicates the \( i \)-th character of \( w \))

\( \Sigma = 02146 \in L \)

\[
\begin{array}{c}
58108554421 \ldots \\
\uparrow \\
011234567890... \\
\uparrow \\
w_7 = 4 \\
w_3 = w_{7-4} = 0
\end{array}
\]

**NTM:**

\( Q = \{q_0, f, [q, 0], [q, 1], \ldots, [q, 3]\} \)

\( F = \{f\} \)

\( \Gamma = \{0, 1, \ldots, 3, \#\} \)
Idea for \( N \): scan \( w \) from left to right,
- guess at some \( w_j = i \),
- store \( i \) in CPU register, and
- move \( i \) steps left to find \( 0 \)

Preconditions:
- \( \delta(q_0, 0) = \{(q_0, 0, R)\} \) \hspace{1cm} (since \( w_j > 0 \))
- \( \forall i > 0 : \delta(q_0, i) = \{(q_0, i, R), ([\uparrow, i], i, L)\} \)
  \hspace{1cm} \text{guess}
- \( \forall i \geq 2, \forall x \in \Sigma : \delta([\uparrow, i], x) = \{(\uparrow, i-1), x, L)\}
- \text{accepting: } \delta([\uparrow, 1], 0) = \{(\uparrow, 0, R)\}

Execution traces on input \( w = 103332 \):

\[
q_0, 103332 \rightarrow 1q_0, 03332 \rightarrow 10q_0, 3332 \rightarrow 103q_0, 332 \rightarrow
\]
\[
\vdots \rightarrow 10, [\uparrow, 3], 3332 \rightarrow 1, [\uparrow, 2], 03332 \rightarrow [\uparrow, 1], 103332
\]
\hspace{1cm} \Rightarrow \text{reject}

\[
q_0, 103332 \rightarrow * 10339, 032 \rightarrow 103, [\uparrow, 3], 332 \rightarrow
\]
\[
\vdots \rightarrow 10, [\uparrow, 2], 3332 \rightarrow 1, [\uparrow, 1], 03332 \rightarrow 10f, 3332
\]
\hspace{1cm} \Rightarrow \text{accept}

**Theorem:** Let \( N \) be a NTM. Then there exists a DTM \( D \) s.t.: \( L(D) = L(N) \)

**Proof:**
- Given \( N \) and \( w \), we show how a multi-tape DTM can simulate the execution of \( N \) on input \( w \).
- We can then convert the multi-tape DTM to a single-tape DTM.
Idea for the simulation:

Consider the execution tree of $N$ on $w$

```
ID_0 = q_0 w
   / \  \\
  ID_1 -> ID_2
   /    \  \\
 ID_3   ID_4
```

DTM $D$ will perform a breadth-first search of the execution tree, systematically enumerating the $ID_0$, until it finds an accepting one.

We use two tapes:

- **tape 2:** is for working
- **tape 1:** contains a sequence of $ID$'s of $N$ in BFS order
  - * used to separate two $ID$'s
  - ^ marks next $ID$ to be explored
    - $ID$'s to the left of ^ have been explored
    - $ID$'s to the right of ^ are still to be explored
  - initially, only $ID_0 = q_0 w$ is on the tape
  - we can use multiple tracks for convenience
Algorithm: repeat the following steps

Step 0: examine current ID_{C} (the one after \(^{\hat{}}\)) and read \( q, e \) from it

if \( q \in F \), then accept and halt

Step 1: let \( \delta(q, e) \) have \( k \) possible transitions

- copy \( ID_{C} \) onto tape 2
- make \( k \) new copies of \( ID_{C} \) and place them at the end of tape 1

Step 2: modify the \( k \) copies of \( ID_{C} \) on tape 1 to become the \( k \) possible outcomes of \( \delta(q, e) \) on \( ID_{C} \)

Step 3: move \( \hat{ } \) right past \( ID_{C} \).

clean up tape 2

return to step 0

It is possible to verify:
- the above steps can all be implemented on a DTM
- the construction is correct, i.e. \( w \in L(D) \) iff \( w \in L(N) \)

Evolution of tape 1:

1) \( \hat{ } ID_{0} \)
2) \( \hat{ } ID_{0} * ID_{0} * ID_{0} * ID_{0} * ID_{0} * \)
3) \( \hat{ } ID_{0} * ID_{1} * ID_{2} * ID_{3} * \)
4) \( \hat{ } ID_{0} * ID_{1} * ID_{2} * ID_{3} * \)
5) \( \hat{ } ID_{0} * ID_{1} * ID_{2} * ID_{3} * ID_{4} * ID_{5} * \)
6) \( \hat{ } 1, 1, - ID_{11} * ID_{22} * \)
7) \( \hat{ } ID_{0} * ID_{4} * ID_{2} * ID_{3} * ID_{1} * ID_{2} * ID_{42} * \)
Simulation time:

Let NTM $N$ have running time $T(m)$. What is the running time of $D$?

Let $m$ be the maximum number of non-det. choices for each transition (i.e., the maximum size of $\delta(q, x)$).

Consider execution tree of $N$ on $w$.

Let $t = T(|w|) \Rightarrow$ exec. tree has at most $t$ levels.

Size of the tree is $\leq 1 + m + m^2 + \ldots + m^t = \frac{m^{t+1} - 1}{m - 1} = O(m^t)$.

Thus $D$ has at most $O(m^t)$ iterations of steps 0-3.

Each iteration requires at most $O(m^t)$ steps.

$\Rightarrow$ Total running time is $m O(t)$, i.e. exponential.