Question: Using a counting argument, we have seen that there are functions that cannot be computed (or, in other words, problems that cannot be solved by any algorithm).

How can we exhibit a specific problem of this form?

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 the specific problem?

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" as its first 12 characters?
Consider a solution to HWP:

```
 P
 H
 I
 ?
 ?
 input

 Does such a program H exist?
 - we could scan P for println 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 $n>3$: 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.C.T. is false, otherwise $P_1$ loops forever.
If we could solve HWP, we would also have proven or disproven F.L.T.

This would be too nice! Where is the problem?

Theorem: There is no Java program $H$ that decides HWP.

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

Consider $H$:

\[ P \xrightarrow{H} \text{"yes"} \]
\[ I \xrightarrow{H} \text{"no"} \]

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

$P \xrightarrow{H_1} \text{yes}$

$I \xrightarrow{H_1} \text{Hello, world}"

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

We modify $H_1$ to $H_2$, which takes only input $P$ and feeds it to $H_1$ as both $P$ and $I$:

Set us consider $H_2(P)$ when $P = H_2$.

\[ H_2(H_2) = \text{"yes"} \Rightarrow P(P) = \text{"Hello, world"} \]
\[ H_2(H_2) = \text{"Hello, world"} \Rightarrow P(P) \neq \text{"Hello, world"} \]

But $P = H_2 \Rightarrow \text{Contradiction} \Rightarrow H_1, H_2$ 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**

**Do-s 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 \((Q, y)\).
Idea: apply modifications to $Q$

1) remove function foo in $Q$ (if present) to $Q_1$.
   $\Rightarrow Q_1$

2) add a dummy function `foo` to $Q_1 \Rightarrow Q_2$

3) modify $Q_2$ to store all its output in some array $A$
   $\Rightarrow Q_3$

4) modify $Q_3$ so that after every println statement
   it checks array $A$ to see if "Hello, world!" has been
   printed. If yes, then call function `foo` $\Rightarrow Q_4$

Note: We can write a Java program that takes as input a
Java source file and modifies it as specified above.

Let $R = Q_4$ and $z = y$

We have by construction:

$Q(y)$ prints "Hello, world!" $\Rightarrow$

$R(z)$ calls function `foo`.

Hence, we can use $F$ which solves foo-problem on $R(z)$
to construct $H$ that solves HWF on $Q(y)$.

Schematically:

\[
(Q, y) \xrightarrow{(\text{construct } (R, z) \text{ from } (Q, y))} (R, z) \xrightarrow{F} \begin{cases} \text{"yes"} \\ \text{"no"} \end{cases}
\]

\[H\]

But since $H$ does not exist, also $F$ cannot exist.

Q.E.D.
Showing undecidability by reduction from undecidable problem

Problem \( P_1 \) taking input \( I_1 \) known to be undecidable
\[ \Rightarrow P_2 \quad \Rightarrow \quad I_2 \] to show undecidable.

Reduction: convert \( I_1 \) to \( I_2 \) such that
\[ P_1(I_1) = \text{"yes"} \iff P_2(I_2) = \text{"yes"} \]

Given solution program \( S_2 \) for \( P_2 \), we could obtain
\[ \Rightarrow S_2 \quad \text{for} \quad P_1 \]

\[ S_1 \]

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:

- \( P \) a problem \( \text{is a language over} \ \Sigma \) (for some finite \( \Sigma \))
  (the strings in the language represent those instances
  of \( P \) for which the answer is "yes")
  \[ \Rightarrow \text{there are uncountably many problems} \]
- \( P \) an algorithm is a string over \( \Sigma' \) (for some finite \( \Sigma' \))
  \[ \Rightarrow \text{there are countably many algorithms} \]
\[ \Rightarrow \text{there must be (uncountably many) problems for which}
\text{there is no algorithm.} \]
Jens (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

We 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:

- Infinite tape
- Read/write head
- Finite state control

Programmed by specifying transitions:
- more 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/stay

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

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

However: a TM can simulate a computer (with a cubic increase in running time — see book 8.6)
Definition \( \mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, \delta_b, F) \)

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

Conditions: \( \Sigma \subseteq \Gamma \), since input is written initially on tape

Initially:
- state \( q_0 \)
  - tape contains \( w \) surrounded by \( \delta_b \)
  - tape head is at the leftmost cell of the input

Transitions:
\[ \delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\} \]

\[ \delta(q, \gamma) = (q', \gamma', d) \]

means that

- if \( M \) is in state \( q \) and tape head is over symbol \( \gamma \),
  - then \( M \) changes state to \( q' \)
    - replaces \( \gamma \) by \( \gamma' \) on the tape
    - moves tape head by one cell in direction \( d \)
      - \( \delta_b \) for \( L \), \( R \), \( S \) for stay in place

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:
- halts 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 or rejects if it loops forever.

Example:

\[
L = \{ w \# w' \mid w, w' \in \{0, 1\}^+, \# \in \{0, 1, \#\}^* \}
\]

Initially:

\[
\begin{array}{cccccccccccc}
0 & \# & 0 & \# & \ldots & 0 & \# & \ldots & 1 & \ldots & 0 & \#
\end{array}
\]

TM idea: remember (on the state) leftmost symbol, and 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.

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

\[
\begin{aligned}
Q &= \{q_0, q_1, \ldots, q_7\} \quad & F &= \{q_7\} \\
\Sigma &= \{0, 1, \#\} \quad & \Gamma &= \{0, 1, \#, \#\}
\end{aligned}
\]

\[
\begin{aligned}
\delta(q_0, 0) &= (q_1, \#, R) \quad & \text{erase 0 and look for matching 0} \\
\delta(q_0, 1) &= (q_1, \#, R) \quad & \text{else if 1, then reject} \\
\delta(q_1, 0) &= (q_1, 0, R) \\
\delta(q_1, 1) &= (q_1, 1, R) \\
\delta(q_1, \#) &= (q_3, \#, R) \\
\delta(q_2, 0) &= (q_2, 0, R) \\
\delta(q_2, 1) &= (q_2, 1, R) \\
\delta(q_2, \#) &= (q_4, \#, R)
\end{aligned}
\]

\[
\text{shift over 0's and 1's, fill \# in found (remembering 0)}
\]

\[
\text{shift over 1's (remembering 1)}
\]
\[ \delta(q_3, #) = (q_3, #, R) \]
\[ \delta(q_3, 0) = (q_3, ., L) \]

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

\[ \delta(q_5, #) = (q_5, ., L) \]
\[ \delta(q_5, 0) = (q_6, 0, L) \]
\[ \delta(q_5, 1) = (q_6, 1, L) \]
\[ \delta(q_5, \$) = (q_7, \$, S) \]

\[ \delta(q_6, 0) = (q_6, 0, L) \]
\[ \delta(q_6, 1) = (q_6, 1, L) \]
\[ \delta(q_6, \$) = (q_0, \$, R) \]

**Note:** If after \( \# \)'s a 0 or a \$ is found, \( M \) halts and rejects any previous ones, replacing 0/1 with 1/0.

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 \$ is found, accept.

More left, skipping 0/1 and \$, end current option.

---

**Transition diagram**

![Transition diagram](image)

- \( \delta(q, x) \) represents \( \delta(q, x) = (q', y, d) \)
Instantaneous description (I.D.) on configuration of a TM describes the current situation of TM and tape.

I.D. = \( \alpha_1, \eta \alpha_2 \) with \( \eta \in Q \)

\[ \alpha_1, \alpha_2 \in \Gamma^* \]

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

Corresponds to:

<table>
<thead>
<tr>
<th>BLANKS</th>
<th>( \alpha_1 )</th>
<th>( \alpha_2 )</th>
<th>BLANKS</th>
</tr>
</thead>
</table>

Let \( ID = \Gamma^* \times Q \times \Gamma^* \) be the set of instantaneous descriptions. We use a relation \( \vdash \in ID \times ID \) to describe the transitions of the TM.

Example:

\[ q_0 \#01 \vdash q_1 \#01 \vdash q_1 \#01 \vdash \]
\[ \vdash \#q_3 \#01 \vdash q_5 \# \# \# \vdash \]
\[ q_5 \# \# \# \vdash q_6 \# \# \# \vdash \]
\[ q_6 \# \# \# \vdash \]
\[ q_9 \# \# \# \vdash q_9 \# \# \# \vdash \text{accept} \]

Note: we can define \( \vdash \) formally, making use of \( S \). 

Exercise 1: Making use of the closure \( \vdash^* \) of \( \vdash \) we can define the language accepted by a TM.

Definition: Let \( M = (Q, \Sigma, \Gamma, \delta, q_0, \#, F) \) be a TM.

Then the language \( L(M) \) accepted by \( M \) is

\[ L(M) = \{ w \in \Sigma^* | q_0 w \vdash^{*} \alpha_1 \eta \alpha_2 \text{ with } \eta \in F \text{ and } \alpha_1, \alpha_2 \in \Gamma^* \} \]
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 test symbols

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

Idea: state names are a tuple of the form

\[ (q_i, D_i, \ldots, D_k) \]

- \( D_i \): each en stored symbol
- \( q_i \): control portion of the state

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

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

\[ Q = \{ [q_i, e] \mid i \in \{0,1\}, \ e \in \{0,1,-\} \} = \{ [q_0, -], [q_0, 0], [q_0, 1], [q_0, -], [q_1, 0], [q_1, 1] \} \]

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

\[ q_0 = [q_0, \#] \quad F = \{ [q_1, -] \} \]

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

- data portion \( e \): \( e \) is the first symbol read.
Transitions:
\[ \delta([q_0, -], a) = ([q_1, e], a, R) \text{, for } a \in \{0, 1\} \]

- M remembers in \([q_0, e]\) that it has read \(a\)

\[ \delta([q_1, O], 0) = ([q_1, O], 1, R) \]

- M moves right as long as it does not see the first symbol

\[ \delta([q_1, 1], O) = ([q_1, 1], O, R) \text{, for } a \in \{0, 1\} \]

- M accepts when it reaches the first \(\Box\)

2) Multiple tracks.

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>*</th>
<th>(\Box)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(a)</td>
<td>(a)</td>
<td>(a)</td>
<td>(c)</td>
</tr>
</tbody>
</table>

The symbols on the tape are \([\frac{0}{a}], [\frac{*}{a}], [\frac{\Box}{a}]\)

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

We first need to find midpoint, and then we can match corresponding symbols.
To find midpoint: we view tape as 2 tracks

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>011</td>
<td>1</td>
<td>011</td>
</tr>
</tbody>
</table>

Hence: \(\Gamma = \{[\frac{0}{a}], [\frac{*}{a}], [\frac{\Box}{a}], [\frac{X}{a}], [\frac{Y}{a}]\}\)

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

\[
\begin{align*}
\delta(q_0, [L^*]) &= (q_1, [L^*], R) & \text{move right till end} \\
\delta(q_1, [L^*]) &= (q_1, [L^*], R) & \text{on first marked symbol} \\
\delta(q_1, [B]) &= (q_2, [B], L) & \text{move rightmost marker one symbol to the left} \\
\delta(q_1, [B]) &= (q_2, [B], L) \\
\delta(q_2, [B]) &= (q_3, [B], L) & \text{move left till end} \\
\delta(q_3, [B]) &= (q_3, [B], L) \\
\delta(q_3, [B]) &= (q_0, [B], R) & \text{on first marked symbol}
\end{align*}
\]

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

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

3) Subroutines / procedure calls

Example: shifting over

Given: \( \text{ID}_1 = \alpha \ q_i \times \beta \) 

Want: \( \text{ID}_2 = \alpha \ \square \ q_i \times \beta \)

for \( \alpha, \beta \in \Gamma^* \)

\( \square \in \Gamma \)

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

E.g. to implement a counter

\[
\begin{align*}
\$0\$ & \rightarrow \$1\$ & \rightarrow \$01\$ & \rightarrow \$01\$ & \rightarrow \$10\$ & \rightarrow \$11\$ & \rightarrow \$011\$ & \rightarrow ... \\
& \rightarrow \$1\$ & \rightarrow \$01\$ & \rightarrow \$011\$ & \rightarrow \$011\$ & \rightarrow ...
\end{align*}
\]

6/12/2006
Procedure call: $\delta(q_i, x) = ([q, x], [\delta], R), \forall x \in \Gamma$

- Remember return state $q_i$, and erased symbol $x$
- State $q_i$ calls procedure

Procedure $\gamma$ for shifting

1) Shift 1 cell to the right

$\delta([q, x], y) = ([q, y], x, R), \forall x, y \in \Gamma$ with $y \neq \#$

2) Till we have reached end of $\beta$

$\delta([q, y], \#) = (q, y, L), \forall y \in \Gamma$

3) Return to calling point by moving left

$\delta(q, y) = (q, y, L), \forall y \neq [\delta]$

4) Exit and return to state $q_i$

$\delta(q_i, [\delta]) = (q_i, \#, R)$

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 heads, 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
- non-determinism

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 behavior of each head independently

$$d(q, x_1, \ldots, x_h) = (q', (y_1, d_1), \ldots, (y_h, d_h))$$

$x_i$: symbol under head $i$

$y_i$: new symbol written by head $i$

$d_i$: direction in which head $i$ moves
To simulate a $k$-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</th>
<th>B</th>
<th>A</th>
<th>C</th>
<th>E</th>
<th>A</th>
<th>Tape 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Head 1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Tape 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Head 2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Tape 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Head 3</td>
</tr>
</tbody>
</table>

Each transition of $M_k$ is simulated by a series of transitions of $M_1$: $\delta(q, x_1, \ldots, x_k) = (q', (y_1, d_1), \ldots, (y_k, d_k))$
- 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 a TM. They do affect its efficiency.

Definition: a 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 \( M \) 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 moves for each transition of \( M_k \)
- each move 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 step, 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: \(\Sigma = \{0, 1, \ldots, 3\}\)

\[L = \{w \in \Sigma^* \mid \text{a 0 appears in positions to the left of some } i \text{ in } w, \text{ with } 0 < i \leq 8\} =\]

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

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

\(\delta_c: 02146 \in L\)

\[
\begin{array}{cccc}
5 & 8 & 4 & 0 \\
8 & 5 & 5 & 4 \\
2 & 1 & 4 & 6 \\
\end{array}
\]

\(w = 01234567890\)

\(w_2 = 4\)

\(w_3 = w_7 = 0\)

NTM \(N\) s.t. \(L(M) = L\)

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

\(F = \{q_f\}\)

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

Transitions:
- \( \delta(q_0, 0) = \{(q_0, 0, R)\} \)  
  (since \( w_j > 1 \))
- \( \forall i > 0 : \delta(q_0, i) = \{(q_0, i, R), ([q, i], i, L)\} \)
  \text{guess}
- \( \forall i \geq 2, \forall x \in \Gamma : \delta([q, i], x) = \{(q, i-1), x, L) \)
- accepting: \( \delta([q, 1], 0) = \{(q, 0, R)\} \)

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

\[
q_0103332 \rightarrow q_003332 \rightarrow 10q_03332 \rightarrow 103q_0332 \rightarrow \\
\rightarrow 10[q, 3]332 \rightarrow 1[q, 2]3332 \rightarrow [q, 1]103332 \rightarrow \text{reject}
\]

\[
q_010332 \rightarrow 10339_032 \rightarrow 1034[q, 3]32 \rightarrow \\
\rightarrow 10[q, 2]3332 \rightarrow 1[q, 1]03332 \rightarrow 1003332 \rightarrow \text{accept}
\]

Theorem: Let \( \mathcal{N} \) be a NTM. Then there exists a DTM \( \mathcal{D} \) s.t.
\[
L(\mathcal{D}) = L(\mathcal{N})
\]

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

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

$ID_0 = q_0 \cdot w$  \[\rightarrow\]  $ID_{21}$  \[\rightarrow\]  $ID_{31}$  \[\rightarrow\]  ...

DTM $D$ will perform a breadth-first search of the execution tree, systematically enumerating the $ID$s, 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 $ID_0 = q_0 \cdot w$ is on the tape
- we can use multiple tracks for convenience
Algorithm: repeat the following steps

Step 0: examine current $D_c$ (the one after *) and read $q, e$ from it
if $q \notin F$, then accept and halt

Step 1: let $S(q, e)$ have $k$ possible transitions
- copy $D_c$ onto tape 2
- make $k$ new copies of $D_c$ and place them at the end of tape 1

Step 2: modify the $k$ copies of $D_c$ on tape 1 to become the $k$ possible outcomes of $S(q, e)$ on $D_c$

Step 3: move * right past $D_c$
clean up tape 2
return to step 0

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

Evolution of phrase 1:
1) * $1D_o$ *
2) * $1D_0`1D_0`1D_0`1D_0`$
3) * $1D_0`1D_1`1D_2`1D_3`$
4) * $1D_0`1D_1`1D_2`1D_3`$
5) * $1D_0`1D_1`1D_2`1D_3`1D_4`1D_5`$
6) * $1D_0`1D_1`1D_2`1D_3`1D_4`1D_5`$
7) * $1D_0`1D_1`1D_2`1D_3`1D_4`1D_5`
Let NTM \( N \) have running time \( T(n) \).

What is the running time of \( D \)?

Let \( m \) be the maximum number of nondet. 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 \( \leq 1 + m + m^2 + \cdots + m^t = \frac{m^{t+1} - 1}{m-1} = O(m^t) \)

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 \( m \cdot O(t) \), i.e. exponential.