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 a 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:

```
\[ P \rightarrow H \rightarrow \text{"yes"} \rightarrow \text{"no"} \rightarrow \text{output} \]
```

Input

Does such a program \( H \) exist?

- We could see \( P \) 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 \( n \geq 3 \): 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 simple 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 Java program H that decides HWP.

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

Consider H:

\[
\begin{array}{c}
P \\
\text{H} \\
I \\
\rightarrow \text{"no" on}
\end{array}
\]

We modify H to H₁, s.t. H₁ prints "Hello, world" instead of "no"

\[
\begin{array}{c}
P \\
\text{H₁} \\
I \\
\rightarrow \text{"yes"}
\end{array}
\]

Hello, world

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

We modify H₁ to H₂, which takes input P and feeds it to H₁ as both P and I:

\[
\begin{array}{c}
P \\
\rightarrow \text{Buffer} \\
\rightarrow \text{H₁} \\
\rightarrow \text{"yes" (when P(P) = "Hello, world")}
\end{array}
\]

\[
\begin{array}{c}
\text{"Hello, world" (when P(P) ≠ "Hello, world")}
\end{array}
\]

Set us consider H₂(P) when P = H₂:

- Suppose H₂(H₂) = "yes" \(\Rightarrow\) P(P) = "Hello, world"
- Suppose H₂(H₂) = "Hello, world" \(\Rightarrow\) P(P) ≠ "Hello, world"

But P = H₂ \(\Rightarrow\) contradiction \(\Rightarrow\) H₁, H₂ 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 \( \text{foo} \) 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 $Q_1$.

2) add a dummy function foo to $Q_1$ to $Q_2$.

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

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 to $Q_4$.

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:

- $Q_4(y)$ prints "Hello, world".
- $R(z)$ calls function foo.

Hence, we can use $F$ that solves foo-circle problem on $R(2)$ to construct $H$ that solves HWP on $Q_4(y)$.

Schematically:

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

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

Reduced $P_2$ to $I_2$ to show undecidable.

Reduction: convert $I_1$ to $I_2$ such that

$P_1(I_1) = \text{"yes"}$ if $P_2(I_2) = \text{"yes"}$

Given solution program $S_2$ for $P_2$, we could obtain

$S_2$ for $P_1$

Since $S_1$ does exist, we obtain that $S_2$ cannot exist 

$\Rightarrow P_2$ is undecidable.

Existence of undecidable problems:

While it is 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")

  $\Rightarrow$ there are uncountably many problems

- An algorithm is a string over $\Sigma'$ (for some finite $\Sigma'$)

  $\Rightarrow$ there are countably many algorithms

  $\Rightarrow$ there must be (uncountably many) problems for which there is no algorithm.
Turing Machines

Java (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:

- infinitely tape
- read/write head
- finite state control

Programmed by specifying transitions:
- move depends on:
  - current state (finite by 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

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

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

Conditions: \( \Sigma \subseteq \Gamma \), since input is written initially on tape.
\( \Gamma = \Gamma - \Sigma \), since the rest of the tape is blank.

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

Transitions:
\( \delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L,R,S\} \)

\( \delta(q, \alpha) = (q', \gamma, d) \) means that

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

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

Acceptance: \( w \) is accepted by TM \( M \) if \( M \), when started with \( w \)
in 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 means 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: \( L = \{ w \#^* w \mid w \in \{0, 1\}^+, \# \in \{0, 1, \#\}^* \} \)

Initially

\[
\ldots \# \ldots w \ldots \# \ldots \# \ldots w \ldots \# \ldots
\]

TM idea: remember (in 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

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

\( Q = \{ q_0, q_1, \ldots, q_7 \} \)

\( \Sigma = \{ 0, 1, \# \} \)

\( \Gamma = \{ 0, 1, \#, \ldots \} \)

\( F = \{ q_7 \} \)

\[
\begin{align*}
\delta(q_0, 0) &= (q_1, \#, R) \quad \text{[These 0 and look for matching 0]} \\
\delta(q_0, 1) &= (q_2, \#, R) \\
\delta(q_1, 0) &= (q_1, 0, R) \\
\delta(q_1, 1) &= (q_1, 1, R) \\
\delta(q_1, \#) &= (q_3, \#, R) \\
\delta(q_2, 0) &= (q_2, 0, R) \\
\delta(q_2, 1) &= (q_2, 1, R) \\
\delta(q_2, \#) &= (q_4, \#, R)
\end{align*}
\]

- \( \ldots \) (remembering 0)

- \( \ldots \) (remembering 1)
\[\delta(q_3, \#) = (q_3, \#, R) \]
\[\delta(q_3, 0) = (q_5, \#, L) \}

Skip over \#'s, look for 0, 1, and replace it by \#.

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

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

As previous ones, 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, \#) = (q_7, \#, Q) \}

Move left skipping \#’s.

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

Move left, skipping 0/1’s and 1/0’s, and restart again.

Transition diagram

\[\delta(q, x) = (q, y, d) \]
Immeasurable description (I.D.) or configuration of $e$ TM

describes the current situation of TM and tape.

$$I.D. = \alpha_1 \cdot q \cdot \alpha_2$$ with $q \in Q$

$$\alpha_1, \alpha_2 \in \Gamma^*$$

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

$$\begin{array}{c}
\text{BLANKS} \\
\alpha_1 \\
\alpha_2 \\
\text{BLANKS}
\end{array}$$

Set $ID = \Gamma^* \times Q \times \Gamma^*$ be the set of instantaneous descriptions.
We use a relation $\delta : M \in ID \times ID$ to describe the transitions of a TM. $M$.
(when $M$ is clear from the context, we abbreviate $\delta$ with $\delta$)

Example:

$$\begin{align*}
q_0 \# 01 & \rightarrow q_1 \# 01 \rightarrow 1 q_1 \# 01 \\
1 \# q_3 \# 01 & \rightarrow 1 q_3 \# 1 \\
q_5 \# 1 \# 1 & \rightarrow q_6 \# 1 \# 1 \\
q_0 \# 1 \# 1 & \rightarrow \ldots \\
q_5 \# 1 \# 1 \# 1 & \rightarrow q_7 \# 1 \# 1 \# 1 \quad \text{accepts}
\end{align*}$$

Note: we can define $\delta$ formally, making use of $S$. [Exercise]

Making use of the closure $\Gamma^*$ of $\Gamma$, we can define the language accepted by a TM

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

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

$$L(M) = \{ w \in \Sigma^* \mid q_0 \# w \xrightarrow{*} \alpha \alpha_1 q \alpha_2 \text{ with } q \in F \text{ and } \alpha_1, \alpha_2 \in \Gamma^* \}$$
Relation \( T_M \in \text{ID} \times \text{ID} \) describes the move of a TM.

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

where \( \text{ID} = \Pi^* \times Q \times \Gamma^* \)

- Let \( S(q, K) = (p, y, L) \) be a leftward move of \( M \)

Then \( K_0 \ldots K_{i-1} q K_i K_{i+1} \ldots K_n \to M K_0 \ldots K_{i-2} p K_i \ldots K_m \ldots K_n \)

Note: the head is now at cell \( i-1 \)

There are two exceptions to this general case

1) If \( i = 1 \), then \( M \) moves to the blank to the left of \( K_1 \)

\[ q K_0 \ldots K_n \to M \uparrow BYK_i \ldots K_n \]

2) If \( i = n \) and \( y = B \), then the symbol \( B \) written over \( K_n \)

is not represented in the resulting ID

\[ K_0 \ldots K_{n-1} q K_n \to M K_0 \ldots K_{n-2} p K_{n-1} \]

Similarly, we can define when \( ID_1 \to_M ID_2 \) for a rightward move \( S(q, K) = (p, y, R) \) of \( M \)

[Scenario]
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.

Those 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 triple of the form [q, D_l, ..., D_k]
   D_l: ...act as stored symbol
   q: ...control portion of the state

Example: TM $M = (Q, \Sigma, \Gamma, s, q_0, \emptyset, F)$ for $L = 01^* + 10^*$
   Idea: M remembers the first symbol and checks that it does not reappear
   $Q = \{ [q_i, e] | i \in \{0, 1\}, \ e \in \{0, 1, \#\} \} =$
   $\{ [q_0, -], [q_0, 0], [q_0, 1], [q_0, -], [q_1, 0], [q_1, 1] \}$
   $\Sigma = \{0, 1\}$
   $\Gamma = \{0, 1, \#\}$
   $q_0 = [q_0, \#]$  $\emptyset = \{ [q_1, -] \}$

Meaning of $[q_i, e]$
- control portion $q_i$:
  \begin{itemize}
  \item $q_0$: M has not yet read its first symbol
  \item $q_1$: M has read its first symbol
  \end{itemize}
- data portion $e$: e is the first symbol read
transitions:

\[ \delta([q_0, z], e) = ([q_1, e], e, R) \], for \( e \in \{0, 1\} \)

- \( M \) remembers in \([q_0, e]\) 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.

\[ \delta([q_1, 1], 0) = ([q_1, 1], 0, R) \] \( M \) accepts when it reaches the first \( y \).

2) Multiple tracks:

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

<table>
<thead>
<tr>
<th>( \Gamma )</th>
<th>0</th>
<th>*</th>
<th>( y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

the symbols on the tape are \([0], [1], [z] \]

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

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

\( \text{used to put markers on symbols} \)

Hence: \( \Gamma = \{ [y], [y], [z], [1], [x], [x] \} \)

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

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

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 \( \ast \) above it, in state \( q_0 \).

3) Subroutines / procedure calls

Example: shifting over

Given: \( ID_1 = \alpha \cdot q_i \cdot \beta \)

Want: \( ID_2 = \alpha \cdot \Box \cdot q_i \cdot \beta \)

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

E.g. to implement a counter

\[
\begin{align*}
\text{@0} \, \rightarrow \, \text{@1} \, \rightarrow \, \text{@0 \text{1}} \, \rightarrow \, \text{@0 \text{10}} \\
\rightarrow \, \text{@1 \text{1}} \, \rightarrow \, \text{@0 \text{11}} \, \rightarrow \, \text{@0 \text{111}} \, \rightarrow \, ...
\end{align*}
\]
Procedure call: \( \delta(q_i, \chi) = ([q, \chi], \emptyset, R), \forall \chi \in \Gamma \)
- remember return state \( q_i \), read erased symbol \( \chi \)
- state \( p \) calls procedure

Procedure \( p \) for shifting

1) shift 1 cell to the right
\( \delta([p, \chi], y) = ([p, y], \chi, R), \forall \chi, y \in \Gamma \) with \( y \neq \emptyset \)

2) till we have reached end of \( \beta \)
\( \delta([p, y], \emptyset) = (q, y, L), \forall y \in \Gamma \)

3) return to calling point by moving left
\( \delta(q, y) = (q, y, L), \forall y \neq [\emptyset] \)

4) exit and return to state \( q_0 \)
\( \delta(q, [\emptyset]) = (q_0, \emptyset, R) \)

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.

**Exercise** Implement a procedure call to copy a string to the end of the input, i.e. given \( \Delta_1 = \alpha \beta q_i \beta \beta \gamma \)
we want \( \Delta_2 = \alpha \beta q_i \beta \beta \gamma \beta \beta \beta \)
Example of computation for shifting over

State

\[ q_0 \rightarrow 1 \ 0 \ 0 \ 0 \ 0 \ a \ b \ c \ y \]

we want to place 0 after the 0's

\[ \delta(q_0, a) = ([q, a], [8], R) \]

\[ \delta([q, a], b) = ([q, b], a, R) \]

\[ \delta([q, b], c) = ([q, c], b, R) \]

\[ \delta([q, c], z) = ([q, c], z, \text{L}) \]

\[ \delta(q_1, x) = (q_1, x, \text{L}) \]

\[ \delta(q_1, [8]) = (q_2, \text{D, R}) \]

\[ q_2 \rightarrow 1 \ 0 \ 0 \ 0 \ 0 \ a \ b \ c \ y \]

\[ \delta(q_2, [8]) = (q_3, \text{D, R}) \]

\[ q_3 \rightarrow 1 \ 0 \ 0 \ 0 \ 0 \ a \ b \ c \ y \]
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

\[ \delta(q, x_1, \ldots, x_h) = (q_1, (y_1, d_1), \ldots, (y_h, d_h)) \]

\( x_i \ldots \) symbol under head \( i \)
\( y_i \ldots \) new symbol written by head \( i \)
\( d_i \ldots \) 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 $M_1$:
- one track of $M_1$ to store tape content
- one track of $M_1$ to mark head position with $*$

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

Each transition of $M_k$ is simulated by a series of transitions of $M_1$: $\delta(q, \kappa_1, \ldots, \kappa_k) = (p_1, (y_1, d_1), \ldots, (y_k, d_k))$

- Start at leftmost head-position marker
- Sweep right and remember in appropriate "CPU registers" the symbols $\kappa_i$ under each head (note: there are exactly $k$, and hence finitely many)
- Knowing all $\kappa_i$'s, sweep left, change each $\kappa_i$ 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)

**Exercise**: Provide the details of the construction to convert a 2-tape TM to a 1-tape TM.
Simulation speed:

Note: enhancements do not affect the expressive power of $e$ TM
- they do affect its efficiency

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

Note: $T(m)$ could be infinite

Theorem: If $M_k$ has running time $T(m)$, then $M_1$ will simulate it with running time $O(T(m)^2)$.

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

Consider $M_1$:
- makes two sweeps for each transition of $M_k$
- each sweep takes at most $O(T(m))$
- number of transitions of $M_k$ is $\leq T(m)$

It follows that the total running time is $O(T(m)^2)$. 
2) Non-deterministic TMs (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.

Sample: $\Sigma = \{0, 1, \ldots, 3\}$

$L = \{w \in \Sigma^* \mid x \in \Sigma \text{ appears } n \text{ times to the left of some } y \text{ in } w, \\
\text{with } 0 < i \leq 3\} = \\
\{w \in \Sigma^* \mid \exists j > 0 \text{ s.t. } w_{i-j} - w_j = 0\}$

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

Ex: 02146 $\in L$

58108554421 ...
01234367880 ...
$w_2=4$
$w_5 = w_{5+4} = 0$

NTM $N: \delta, S(M) = L$

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

$F = \{f\}$

$\Gamma = \{0, 1, \ldots, 3, \#\}$
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

Transitions:
- \( \delta(q_0, 0) = \{(q_0, 0, R)\} \) (since \( w_j > 1 \))
- \( \forall i > 0: \delta(q_0, i) = \{(q_0, i, R), ([q, i], i, L)\} \)

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

Execution traces on input \( w = 103332 \)

\[
q_0103332 \rightarrow q_00103332 \leftarrow 10q_03332 \rightarrow 103q_0332 \leftarrow \\
\quad 1001332 \leftarrow 10320332 \leftarrow 101103332 \\
= \text{reject}
\]

\[
q_0103332 \rightarrow 1033932 \rightarrow 1033932 \leftarrow 10333[1, 3]332 \leftarrow \\
\quad 10[1, 2]3332 \leftarrow 1[1, 1]03332 \leftarrow 10[1, 1]3332 \\
= \text{accept}
\]

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 \( w \)

\[
\begin{align*}
ID_0 &= \text{q}_0 \cdot w \\
ID_1 &\rightarrow D_{10} \\
ID_2 &\rightarrow D_{21} \\
ID_3 &\rightarrow D_{31} \\
\end{align*}
\]

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 \( ID_0 = \text{q}_0 \cdot w \) 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 \( \ast \)) and read \( q, \varepsilon \) from it

- If \( q \in F \), then accept and halt

Step 1: let \( \delta(q, \varepsilon) \) 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 \( \delta(q, \varepsilon) \) on \( ID_c \)

Step 3: move \( \ast \) 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) \( \ast ID_0 \ast \)
2) \( \ast ID_0 \ast ID_0 \ast ID_0 \ast ID_0 \ast \)
3) \( \ast ID_0 \ast ID_1 \ast ID_2 \ast ID_3 \ast \)
4) \( \ast ID_0 \ast ID_1 \ast ID_2 \ast ID_3 \ast \)
5) \( \ast ID_0 \ast ID_1 \ast ID_2 \ast ID_3 \ast ID_4 \ast ID_5 \ast \)
6) \( \ast \ast ID_9 \ast ID_{12} \ast \)
7) \( \ast ID_0 \ast ID_4 \ast ID_2 \ast ID_3 \ast ID_11 \ast ID_{12} \ast \)
Simulation time:

Let NTM \( N \) have running time \( T(n) \).

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 \( \leq 1 + m + m^2 + \ldots + m^t \)

\[ \leq \frac{m^{t+1} - 1}{m - 1} = O(m^t) \]

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.