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, HelloWorld:

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

The first 12 characters output by HelloWorld 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  |  "yes"
  |   |  "no"
  |   |  output
   I  
   ?
input  
   ?
  gene program
```

Does such a program H exist?
- we could scan 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 gene program \( P_1 \) that:

1) reads input \( n \)

2) for all possible \( x, y, z \) do
   if \( (x^n + y^n = 2^n) \)
       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 $H$:

- $P \rightarrow H \rightarrow \text{"yes"}
- I \rightarrow H \rightarrow \text{"no"}

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

- $P \rightarrow H_1 \rightarrow \text{"yes"}
- I \rightarrow H_1 \rightarrow \text{"Hello, world"}

(Note: we have to modify the question 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 $I$:

- $P \rightarrow \text{Buffer} \rightarrow H_1 \rightarrow \text{"yes" (when $P(P) =$ "Hello, world")}
- \quad \quad \rightarrow \text{"Hello, world" (when $P(P) \neq \text{"Hello, world"}$)}

Now program reads in $P$, stores it in a String and prints it twice to $H_1$.

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

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

But $P = H_2 \Rightarrow \text{Contradiction} \Rightarrow H_1, H_1, H_2$ 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 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 get $Q_1$.

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

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

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

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

Let $R = Q_4$ and $Z = y$.

We have by construction:

$Q(y)$ prints "Hello, world" $\iff$ $R(z)$ calls function foo.

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

Schematically:

\[
\begin{array}{c}
(Q, y) \\
\mapsto \begin{array}{c}
\text{(Construct } (R, z) \text{ from } (Q, y)) \\
\end{array} \\
\mapsto \begin{array}{c}
F \\
\end{array} \\
\mapsto \begin{array}{c}
\text{"yes" or} \\
\text{"no"} \\
\end{array}
\end{array}
\]

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

\[ \Rightarrow \quad P_2 \quad \Rightarrow \quad I_2 \quad \text{to show undecidable.} \]

Reduction: convert \( I_1 \) to \( I_2 \) such that

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

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

\[ \Rightarrow \quad S_2 \quad \text{for} \quad P_1 \]

\[ S_1 \]

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

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

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")

\[ \Rightarrow \quad \text{there are uncountably many problems} \]

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

\[ \Rightarrow \quad \text{there are countably many algorithms} \]

\[ \Rightarrow \quad \text{there must be (uncountably many) problems for which there is no algorithm.} \]
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:

- 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 books 8.6)
Definition $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, \beta, F)$

- $Q$: set of states (finite)
- $q_0 \in Q$: initial state
- $\Sigma$: input alphabet (finite)
- $\Gamma$: tape alphabet (finite)
- $F \subseteq Q$: final states
- $\beta \in \Gamma$: blank symbol
- $\delta: \Sigma \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\}$

- $\delta(q, \xi) = (q', \eta, d)$ means that
  - if $M$ is in state $q$ and tape head is over symbol $\xi$
    - then $M$ changes state to $q'$
    - replaces $\xi$ by $\eta$ 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 $\delta(q, \xi)$ we have at most one move
- ($\delta(q, \xi)$ 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: $w$ is not accepted by TM $M$ if $M$, when started with $w$ on the tape, never halts.

Rejection: $w$ is not accepted by TM $M$ if $M$, when started with $w$ on the tape, 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: \( L = \{ w \#^* \mid w \in \{0, 1\}^+, \# \in \{0, 1, \#\}^* \} \)

Initially

\[
\begin{array}{cccccccccccc}
\& \& \& \& \& \& \& \& \& \& \& \& \\
\# \& \# \& \# \& \# \& \# \& \# \& \# \& \# \& \# \& \# \& \# \\
\end{array}
\]

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.

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

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

\[
\begin{align*}
\delta(q_0, 0) &= (q_1, \#, R) \quad \text{erase 0 and look for matching 0} \\
\delta(q_0, 1) &= (q_2, \#, R) \quad \text{... 1 ...} \\
\delta(q_1, 0) &= (q_2, 0, R) \quad \text{shift over 0's and 1's,} \\
\delta(q_1, 1) &= (q_1, 1, R) \quad \text{kill \# is found (remembering 0)} \\
\delta(q_1, \#) &= (q_3, \#, R) \quad \text{... 1 ...} \\
\delta(q_2, 0) &= (q_2, 0, R) \quad \text{(remembering 0)} \\
\delta(q_2, 1) &= (q_2, 1, R) \quad \text{(remembering 1)} \\
\delta(q_3, 0) &= (q_7, 0, R) \\
\delta(q_3, 1) &= (q_7, 1, R) \\
\delta(q_7, \#) &= (q_7, \#, R)
\end{align*}
\]
\( \delta(q_3, \#) = (q_3, \#, R) \) \\
\( \delta(q_3, 0) = (q_5, \#, L) \)  \\
Skip over \#'s, look for 0, and replace it by \#.
Note: if after \#'s a 1 or a \$ is found, \( M \) halts and rejects.

\( \delta(q_4, \#) = (q_4, \#, R) \) \\
\( \delta(q_4, 0) = (q_5, \#, L) \)  \\
As previous ones, replacing 0/1 with 1/0.

\( \delta(q_5, \#) = (q_5, \#, L) \) \\
\( \delta(q_5, 0) = (q_6, 0, L) \) \\
\( \delta(q_5, 1) = (q_6, 1, L) \) \\
\( \delta(q_5, \$) = (q_7, \$, 5) \)  \\
Move 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.

\( \delta(q_6, 0) = (q_6, 0, L) \) \\
\( \delta(q_6, 1) = (q_6, 1, L) \) \\
\( \delta(q_6, \$) = (q_0, \$, R) \)  \\
More left, skipping 0/1 and 1/0, and restart again.

**Transition Diagram**

![Transition Diagram](image)

\( \delta(q, x) = (q', y, d) \) represents \( \delta(q, x) = (q', y, d) \)
Suddenaneous description (I.D.) on configuration of a TM

describes the current minition of TM and tape.

I.D. = \( \alpha_1 \# \alpha_2 \) with \( q \in Q \)

\( \alpha_1, \alpha_2 \in \Gamma^* \)

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

Corresponds to

\[
\begin{array}{c|c|c|c}
\text{BLANKS} & \alpha_1 & \alpha_2 & \text{BLANKS} \\
\end{array}
\]

Set \( ID = \Gamma^* \times Q \times \Gamma^* \) be the set of instantaneous descriptions.

We use e relation \( t_M \in ID \times ID \) to describe the transitions of a TM \( M \). (when \( M \) is clear from the context, we abbreviate \( t_M \) with \( t \))

Example:

\[
\begin{align*}
q_0 \#01 & \rightarrow q_1 \#01 \rightarrow 1q_1 \#01 \rightarrow \\
& \rightarrow 1\#q_3 \#01 \rightarrow 1q_5 \#\#1 \rightarrow \\
& \rightarrow q_5 \#\#1 \rightarrow q_6 1\#\#1 \rightarrow \\
& \rightarrow q_0 1\#\#1 \rightarrow \cdots \rightarrow \\
& \rightarrow q_5 \#\#\# \rightarrow q_7 \#\#\# \rightarrow \text{accepts}
\end{align*}
\]

Note: we can define \( t_M \) formally, making use of \( \delta \). [See next page]

Making use of the closure \( t^* \) of \( t \) 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

\[
L(M) = \{ w \in \Sigma^* \mid q_0 w \xrightarrow{t^*} \alpha_1 \alpha_2 \text{ with } q \in F \text{ and } \alpha_1, \alpha_2 \in \Gamma^* \}
\]
Relation \( I_m \subseteq ID \times ID \) describes the move of a TM
\[ M = (Q, \Sigma, \Gamma, \delta, q_0, B, F) \]
where \( ID = \Gamma^* \times Q \times \Gamma^* \)

- Let \( \delta(q, X) = (q', y, L) \) be a leftward move of \( M \)

\[ X_i \ldots X_{i-1} q X \overset{r}{\rightarrow} X_i \ldots X_{i-1} \quad q \overset{L}{\rightarrow} X_i \ldots X_{i+1} \]

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 \( X \)
\[ q X X_2 \ldots X_m \overset{L}{\rightarrow} q^X X_2 \ldots X_m \]

2) if \( i = m \) and \( X = B \), then the symbol \( B \) written over \( X \) is not represented in the resulting ID
\[ X_1 \ldots X_{m-1} q X \overset{L}{\rightarrow} X_1 \ldots X_{m-2} p X_{m-1} \]

Similarly, we can define when \( ID_1, \delta, ID_2 \) for a rightward move \( \delta(q, X) = (q', y, R) \) of \( M \)

[exercise]
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 tuple of the form

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

\( D_i \) : each as stored symbol
\( q \) : control portion of the state

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

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

\[ Q = \{ (q_i, a) \mid i \in \{0, 1\}, a \in \{0, 1, \_\} \} = \]
\[ \{ (q_0, \_), (q_0, 0), (q_0, 1), (q_1, \_), (q_0, 0), (q_1, 1) \} \]

\[ \Sigma = \{ 0, 1 \} \]
\[ \Gamma = \{ 0, 1, \_\} \]
\[ q_0 = (q_0, 0) \]
\[ F = \{ (q_1, \_) \} \]

Meaning of \( (q_i, a) \):
- control portion \( q_i \):
  \( q_0 \) : \( M \) has not yet read its first symbol
  \( q_1 \) : \( M \) has read its first symbol
- state portion \( a \) : \( a \) is the first symbol read.
transitions:
\[ \delta([q_0, \varepsilon], a) = ([q_1, a], a, R), \text{ for } a \in \{0, 1\} \]

...M remembers in \([q_0, \varepsilon]\) that it has read \(a\)

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

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

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

...M accepts when it reaches the first \(\varepsilon\)

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>(\varepsilon)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(a)</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

the symbols on the tape are \([0], [0], [y]\)

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

\[ 0 \ 1 \ 1 \ 0 \ 1 \ 1 \]

\(=\) used to put markers on symbols

Hence: \(\Gamma = \{[0], [0], [y], [x], [x], [x]\}\)

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

\[
\begin{align*}
\delta(q_0, [\ast_i]) &= (q_1, [\ast_i], R) \quad \text{move right till and on first marked symbol} \\
\delta(q_1, [\ast_i]) &= (q_1, [\ast_i], R) \\
\delta(q_1, [\ast_i]) &= (q_2, [\ast_i], L) \\
\delta(q_1, [\ast_i]) &= (q_2, [\ast_i], L) \\
\delta(q_2, [\ast_i]) &= (q_3, [\ast_i], L) \\
\delta(q_3, [\ast_i]) &= (q_3, [\ast_i], L) \\
\delta(q_3, [\ast_i]) &= (q_0, [\ast_i], R) \quad \text{move left till and on first marked symbol}
\end{align*}
\]

Note: we have each of the above for \(i \in \{0, 1\}\) at the end: head is on first symbol of second \(w\), with \(\ast\) above it, in state \(q_0\).

3) Subroutines / procedure calls

Example: shifting over

given: \(1D_1 = \alpha q_i \kappa \beta\)  
want: \(1D_2 = \alpha \square q_i \kappa \beta\)

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

\(\square\) to implement a counter

\(80\$ \rightarrow 81\$ \rightarrow \square81\$ \rightarrow 801\$ \rightarrow 810\$ \rightarrow \)
\(\rightarrow 811\$ \rightarrow \square811\$ \rightarrow 8011\$ \rightarrow \ldots\)
Procedure call: \( \delta(q, x) = \left( ([q, x], \left[ \frac{y}{y} \right], R) \right), \forall x \in \Gamma \)

- remember return state \( q_0 \), and crossed symbol \( x \)
- state \( q \) cells procedure

Procedure \( \eta \) for shifting

1) shift 1 cell to the right
\[ \delta([q, x], y) = ([q, y], x, R) \] \( \forall x, y \in \Gamma \) with \( y \neq \frac{y}{y} \)

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

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

4) exit and return to state \( q_0 \)
\[ \delta(q, [\frac{y}{y}]) = (q, [\frac{y}{y}], R) \]

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 \( I \_D_1 = \alpha \frac{y}{y} q_1 \frac{b}{b} y \)
we want \( I \_D_2 = \alpha \frac{y}{y} q_1 \frac{b}{b} y \frac{b}{b} y \frac{b}{b} \)
Example of computation for shifting over state

\[ q_i \quad \begin{array}{c|c|c|c|c|c|c|c} & 0 & 0 & 0 & a & b & c & \vdots \\ \hline \end{array} \]

we want to place \( \Box \) after the 0's

\[ \delta(q_i, a) = ([q, a], \Box, R) \]

\[ [q, a] \quad \begin{array}{c|c|c|c|c|c|c|c} & 0 & 0 & 0 & 0 & a & b & c & \vdots \end{array} \]

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

\[ [q, b] \quad \begin{array}{c|c|c|c|c|c|c|c} & 0 & 0 & 0 & 0 & 0 & a & b & c & \vdots \end{array} \]

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

\[ [q, c] \quad \begin{array}{c|c|c|c|c|c|c|c} & 0 & 0 & 0 & 0 & 0 & 0 & a & b & c & \vdots \end{array} \]

\[ \delta([q, c], \Box) = (q_i, c, L) \]

\[ q_i \quad \begin{array}{c|c|c|c|c|c|c|c} & 0 & 0 & 0 & 0 & 0 & 0 & a & b & c & \vdots \end{array} \]

\[ \delta(q_i, \Box) = (q_i, \Box, L) \]

\[ q_i \quad \begin{array}{c|c|c|c|c|c|c|c} & 0 & 0 & 0 & 0 & \Box & a & b & c & \vdots \end{array} \]

\[ \delta(q_i, [\Box]) = (q_i, 0, R) \]

\[ q_i \quad \begin{array}{c|c|c|c|c|c|c|c} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & a & b & c & \vdots \end{array} \]
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 \( v \) is on tape 1 with tape head on the leftmost symbol. Other tapes are all blank.

Transitions: specify behaviour of each head independently

\[ s(q_i, x_i, \ldots, x_k) = (q', (y_1, d_1), \ldots, (y_k, d_k)) \]
- \( x_i \) ... symbol under head \( i \)
- \( y_i \) ... new symbol written to head \( i \)
- \( d_i \) ... direction in which head \( i \) moves
To simulate a $k$-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 $*$

<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>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>q</td>
<td>b</td>
<td>q</td>
<td>b</td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tape 1
Head 1
Tape 2
Head 2
Tape 3
Head 3

Each transition of $M_k$ is simulated by a series of transitions of $M_1$: $\delta(q_i, x_i, \ldots, x_k) = (p, (y_i, d_i), \ldots, (y_k, d_k))$
- 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)
- Emulating 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)

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 a 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 $x$ of length $m$.
- $M_k$ runs at most $T(m)$ time on $x$.
- 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) Nondeterministic 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.

$$\delta(q, x) = \{(r_1, y_1, d_1), \ldots, (r_k, y_k, d_k)\}$$

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

$L = \{w \in \Sigma^* \mid a \leq b \}$

$L = \{w \in \Sigma^* \mid \exists j \geq 0 \text{ s.t. } w_{j-4} - w_3 = 0\}$

(Indicates the $i$th character of $w$)

Ex: 02146 $\notin L$

581 0855 4427
0 1 2 3 4 5 6 7 8 9 0
$w_2 = 4$
$w_3 = 4$

NTM $\mathcal{N}$ s.t. $L(\mathcal{N}) = L$

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

$F = \{f\}$

$\Gamma = \{0, 1, \ldots, 3, \#\}$
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 \( w_j > 1 \))
- \( \forall i > 0 : \delta(q_0, i) = \{(q_0, i, R), ([q, i], i, L)\} \)
- \( \forall i \geq 2, \forall x \in \Gamma : \delta([q, i], x) = \{[q, i-1], x, L\} \)
- accepting: \( \delta([q, 1], 0) = \{(q, 0, R)\} \)

Execution traces on input \( w = 103332 \)

\[
q_0 \rightarrow 103332 \rightarrow 1q_0.03332 \rightarrow 10q_0.3332 \rightarrow 103q_0.332 \rightarrow 10[1,3]3332 \rightarrow 1[1,2]03332 \rightarrow [q,1]103332
\]

\( \Rightarrow \) reject

\[
q_0 \rightarrow 103332 \rightarrow \ast \rightarrow 1033 Q_{0.32} \rightarrow 103 [q, 3] 332 \rightarrow 10 [q, 2] 3332 \rightarrow 1 [q, 1] 03332 \rightarrow 10 f 3332
\]

\( \Rightarrow \) accept

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

**Proof:** Given \( N \), we know how to construct a multi-tape DTM that can simulate the execution of \( N \) on some 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 some input \( w \)

\[
\begin{align*}
ID_0 &= q_0 w \\
ID_1 &\rightarrow ID_{a1} \\
ID_2 &\rightarrow D_{21} \\
ID_3 &\rightarrow ID_{31} \\
ID_4 &\rightarrow ID_{41} \\
ID_5 &\rightarrow ID_{51} \\
\end{align*}
\]

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 \( ID_0 \)'s of \( N \) in BFS order
  
  - \( \ast \) used to separate two \( ID_0 \)’s
  
  - \( \ast \) marks next \( ID_0 \) to be explored
  
  - \( ID_0 \)’s to the left of \( \ast \) have been explored
  
  - \( ID_0 \)’s to the right of \( \ast \) are still to be explored

- Initially, only \( ID_0 = q_0 w \) is on the tape
- We can use multiple tracks for convenience

"BFS order" means that the sequence of \( ID_0 \)’s is managed as a queue:

- \( \ast \) determines the beginning of the queue
- The end of tape 1 represents the end of the queue
Algorithm: repeat the following steps

step 0: examine current IDc (the one after \( \hat{\text{ID}} \)) and read q, ε from it

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

step 1: let \( \delta(q, ε) \) have \( k \) possible transitions

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

step 2: modify the \( k \) copies of IDc on tape 1 to become the \( k \) possible outcomes of \( \delta(q, ε) \) on IDc

step 3: move \( \hat{\text{ID}} \) right past IDc

- clear 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 tape 1:

1) \( \hat{\text{ID}} \text{D}_0 \text{ID} \)
2) \( \hat{\text{ID}} \text{ID}_0 \text{ID}_0 \text{ID}_0 \text{ID}_0 \text{ID}_0 \text{ID} \)
3) \( \hat{\text{ID}} \text{ID}_0 \text{ID}_1 \text{ID}_2 \text{ID}_3 \text{ID} \)
4) \( \text{ID}_0 \text{ID}_1 \text{ID}_2 \text{ID}_3 \text{ID} \)
5) \( \text{ID}_0 \text{ID}_1 \text{ID}_2 \text{ID}_3 \text{ID}_4 \text{ID}_5 \text{ID}_6 \text{ID} \)
6) \( \text{ID}_0 \text{ID}_1 \text{ID}_2 \text{ID}_3 \text{ID}_4 \text{ID}_5 \text{ID}_6 \text{ID} \)
7) \( \text{ID}_0 \text{ID}_1 \text{ID}_2 \text{ID}_3 \text{ID}_4 \text{ID}_5 \text{ID}_6 \text{ID}_7 \text{ID}_8 \text{ID}_9 \text{ID} \)
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-delet. choices for each transition (i.e., the maximum size of \( S(q,x) \)).

Consider execution tree of \( N \) on \( w \).

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

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

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

Each iteration requires at most \( O(m^k) \) steps.

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