Models of computation

Let's say that we have two programs A and B. If they behave in the same way on all input it is natural to say that they are equivalent.

A model of computation is an abstract and usually simplified way of describing algorithms. The idea is that, given any algorithm A in any programming language or any other way of presenting algorithms, there should be an algorithm A' in the computational model such that A and A' are equivalent.

Even if two algorithms are equivalent they can have different running times.

A model of computation will be useful when we:

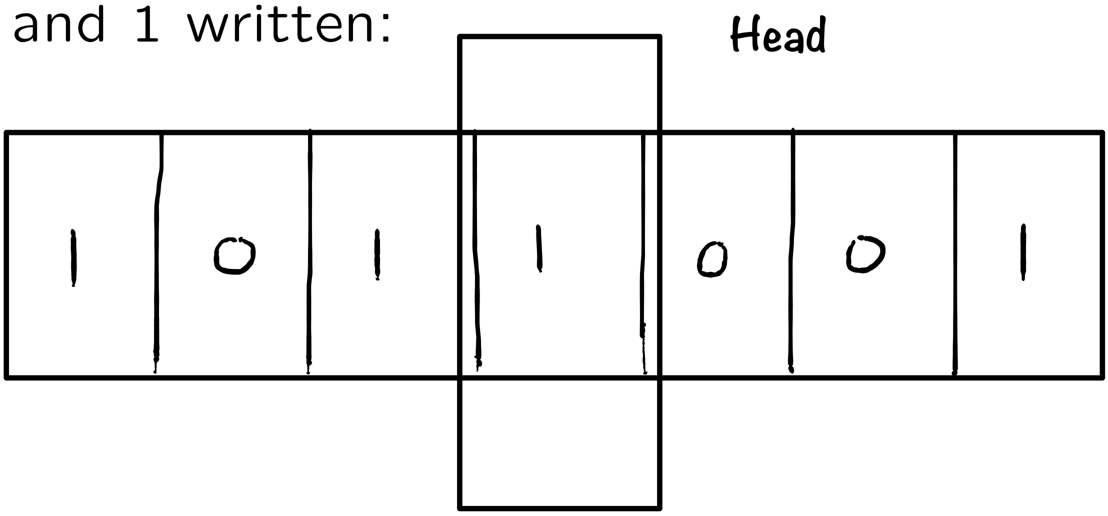
* Want to define exactly what the complexity for an algorithm is.
* Want to find the limits for what algorithms can do.
* Prove Cook's theorem.

One of the oldest but still most useful models of computation is the Turing Machine.

The Turing Machine

We will use a very simplified model of computation. It's the *Turing Machine*.

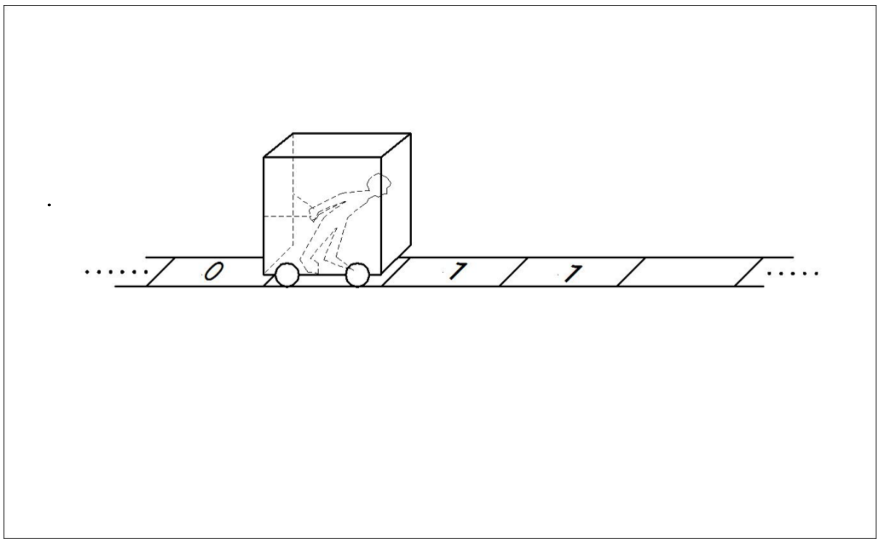
We will consider data as a semi-finite tape with 0 and 1 written:

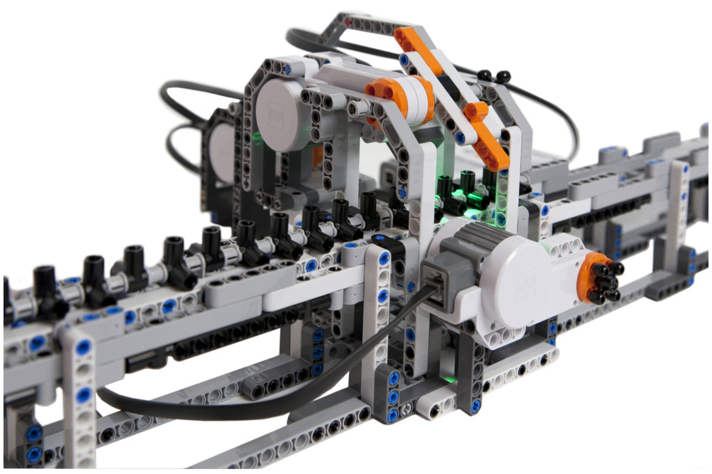


Reading and writing can be done one digit at a time. The "Head" can be moved just one step to the right or left at a time.

The logic tells us how the head should be moved and what should be written on the tape. The logic consists of a finite set of states and a finite set of transition rules.

Turing Machines





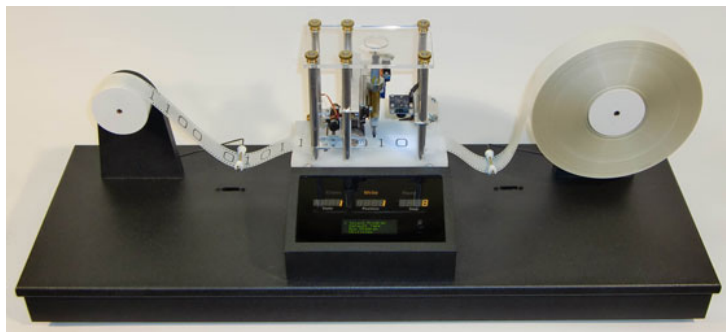
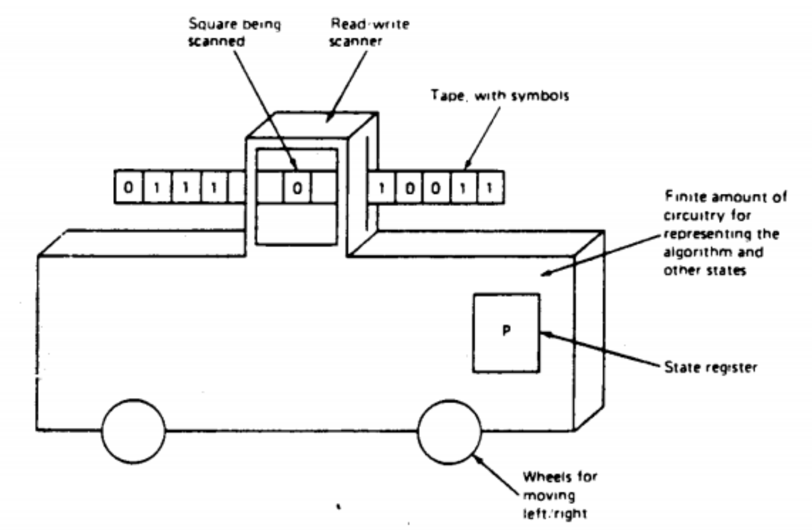
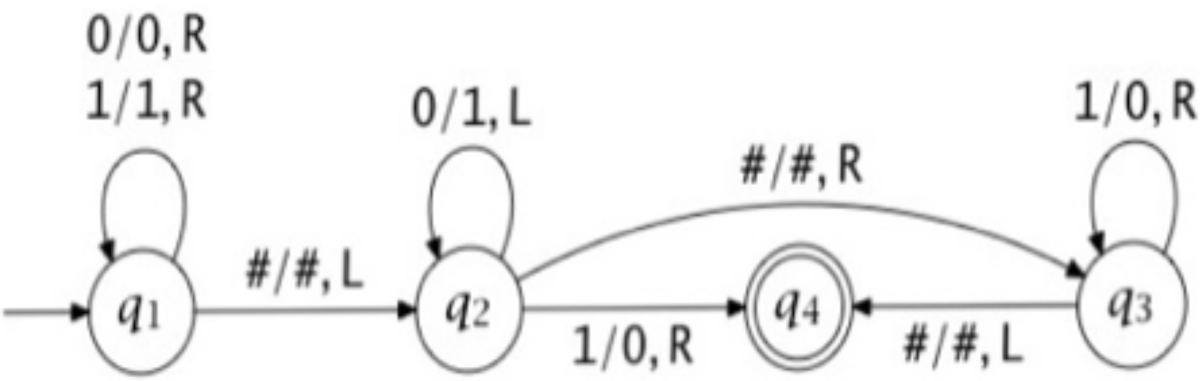


Figure B-2 An imaginary, physical Turing machine.



Example of a Turing Machine

The following TM reads the number x on binary form from the tape and changes it to max$(x-1,0)$.



Notation:

- Circles correspond to states
- Dubble circles correspond to accepting states
- Arrows indicates transition rules:
- $a/b, L$ means "if the head reads a, do the transition, write b and move the head one s to the left"
  (in $a/b, R$ R means move to the right)

The arrow with no starting node indicates the state the machine starts in.

Ex.: <Från OH>

Körning med #1010#

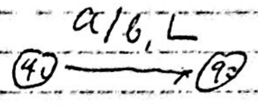
$\#\underset{q_1}{\underline{1}}010\# \rightarrow \#1\underset{q_1}{\underline{0}}10\# \rightarrow \#10\underset{q_1}{\underline{1}}0\# \rightarrow \#101\underset{q_1}{\underline{0}}\# \rightarrow$

$\rightarrow \#1010\underset{q_1}{\underline{\#}} \rightarrow \#101\underset{q_2}{\underline{0}}\# \rightarrow \#10\underset{q_2}{\underline{1}}1\# \rightarrow \#10\underset{q_1}{\underline{0}}1\#$

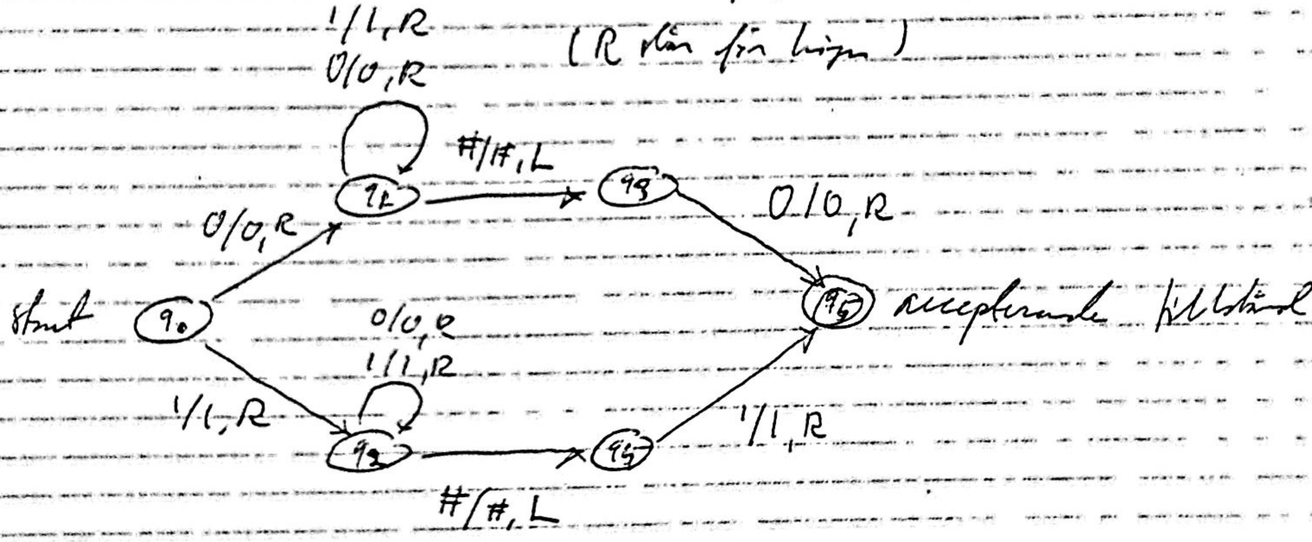
Ex: Maskin som avgör om en sträng börjar och slutar med samma siffra.

Tre symboler 0, 1, # (blankt)

6 tillstånd som ges som noder i graf.



Betyder: Om läshuvudet står i tillstånd q_1 och läser a, gå över i tillstånd q_2, skriv b och flytta 1 steg till vänster. (R står för höger)



Ex: #1001#

$\#\underline{1}001\#$ (q_0) → $\#1\underline{0}01\#$ (q_2) → $\#10\underline{0}1\#$ (q_2) → $\#100\underline{1}\#$ (q_2)

→ $\#1001\underline{\#}$ (q_2) → $\#100\underline{1}\#$ (q_4) → $\#1001\underline{\#}$ (q_5)

Om T stannar i accepterande tillstånd är svaret Ja.

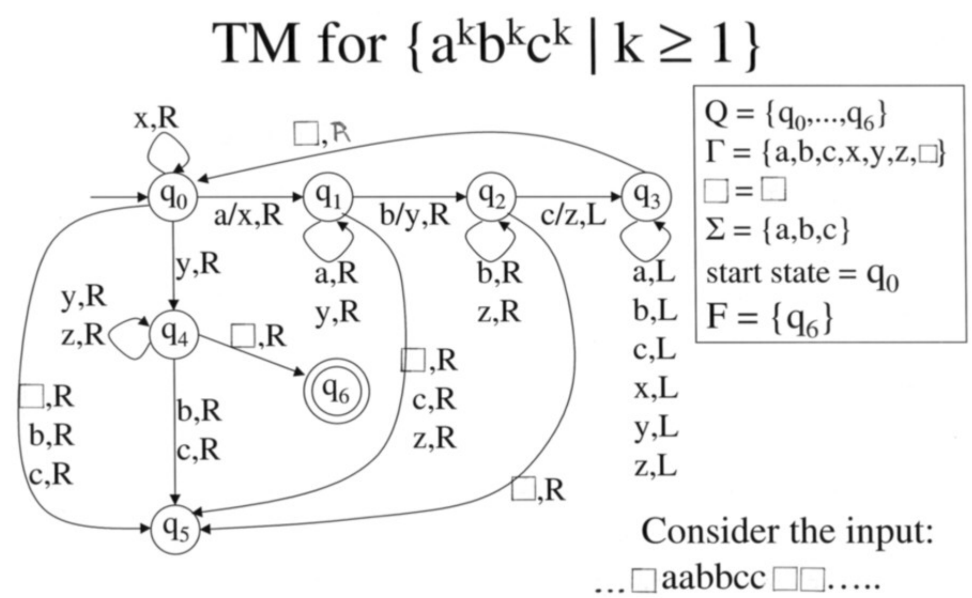
Annars Nej.

Rules for the Turing Machine

- The machine starts in the starting state.
- At start the head reads the first symbol to the left in the input string. The input is marked off by empty positions (indicated by #).
- There must not be several different transitions from the same state reading the same symbol (determinism).
- If the machine gets into an accepting state the computation ends and the machine returns ''Yes''.
- If the machine gets into a state and reads a symbol with no matching transition the computation ends and the machine returns ''No''.

The previous rules describe computations when the answer is yes/no. Turing Machines can do other computations as well. The first example shows this. (The algorithm that computes max(x-1,0).) This is an algorithm of the form A(n) = m, where n and m are integers. As we have seen, Turing Machines can handle them in a rather natural way.

A more advanced Turing Machine



Formal description

A Turing Machine is defined by

- The alphabet Σ (must be finite)
- The set Q of states (must be finite)
- The start state $q_0 \in Q$
- The set $F \subseteq Q$ of accepting states
- The transition relation
  $\Delta \subseteq Q \times \Sigma \times Q \times \Sigma \times \{L, R, S\}$

Church's thesis

Any algorithmic problem that can be solved by any program written in any language and run on any computer can be solved by a Turing Machine.

- The Halting Problem is undecidable even for Turing Machines.
- The Turing Machine can be used as a computational model for reasoning about uncomputability.
- The Halting Problem is undecidable in any computational model powerful enough to simulate a Turing Machine.

The computational model RAM is Turing Equivalent as are all modern programming languages.

Equally powerful variants of the Turing Machine

- A different (finite) alphabet.
- Separate tap for output.
- Several different tapes.
- Several different heads.
- Half-infinite tape (infinite in just one direction).

All these variants are equivalent yo to normal Turing Machine in the sense that the running time differ by at most a polynomial factor.

Non-deterministic Turing Machines

- In the non-deterministic case there can be several possible transitions from a state and a given symbol. In that case, the machine makes a non-deterministic choice.

- If there is a sequence of choices leading to an accepting state we say that the machine *accepts*.

- If there is no sequence of choices leading to an accepting state we say that the machine *rejects*.

Non-determinism cont.

Non-deterministic Turing Machines can be used to define NP:

This class contains exactly the problems (or rather their languages) to which there is an non-deterministic TM that accepts in polynomial time.

NP = **N**on-deterministic **P**olynomial time

One believes that non-deterministic machines are more powerful than deterministic ones in the sense that:

$\mathbf{P} \neq \mathbf{NP}$.

Cook's Theorem

Cook's Theorem says that the problem SAT is NP-Complete.

Input to SAT is a propositional logic formula Φ and the problem is to decide if the formula is satisfiable or not.

Proof of Cook's Theorem (Sketch):

SAT $\in$ **NP** since, given an variable assignment, we can check in polynomial time if the formula is satisfied or not.

We must show that SAT Is NP-Hard, i.e. if om $Q' \in$ **NP** then $Q' \leq_P$ SAT.

Since $Q' \in$ **NP** there is a non-deterministic Turing Machine M that accepts the language Q' in at most kn^c steps where n is the number of variables.

Proof idea:
Construct a formula such that it is satisfied if and only if M accepts the input string.

We assume that M has an input tape that is infinite to the right and uses the alphabet $\{0, 1, \#\}$.

We enumerate M:s time steps from 1 to kn^c.
At each time step t the computation is described by

- the position of the head
- the state q
- the content of the tape in positions $1 - kn^c$

In our formula we use the following variables:

x_{qt} $\quad q \in Q,\ 1 \leq t \leq kn^c$
y_{ijt} $\quad i \in \{0, 1, \#\}, 1 \leq j \leq kn^c, 1 \leq t \leq kn^c$
z_{jt} $\quad 1 \leq j \leq kn^c, 1 \leq t \leq kn^c$

Interpretation:

$x_{qt} = 1$ iff M is in state q at time t
$y_{ijt} = 1$ iff the symbol i is in position j at time t
$z_{jt} = 1$ iff the head stands in position j at time t

If there is an accepting computation for $M(a_1, \ldots, a_n)$ running kn^c steps, then this corresponds to:

1. The computation starts with $a_1, \ldots, a_n$
2. x, y, z describes a correct computation
3. The computation ends in an accepting state.

All these constraints can be expressed by a single SAT-Formula of size polynomial in n.

This gives us an reduction $Q \leq_P SAT$ for every NP-Problem Q and this shows that SAT is NP-Complete.

The universal Turing Machine

To every Turing Machine T we can associate the code k(T) of the machine. If we have input x we say that T(x) is the result of the computation whatever form i might have. It is possible to construct a Turing Machine U that take two strings as input such that

U(k(T), x) = T(x) for all Turing Machines T.

This means that U can simulate every Turing Machine.

It little informal, we can write U(T, x) = T(x).