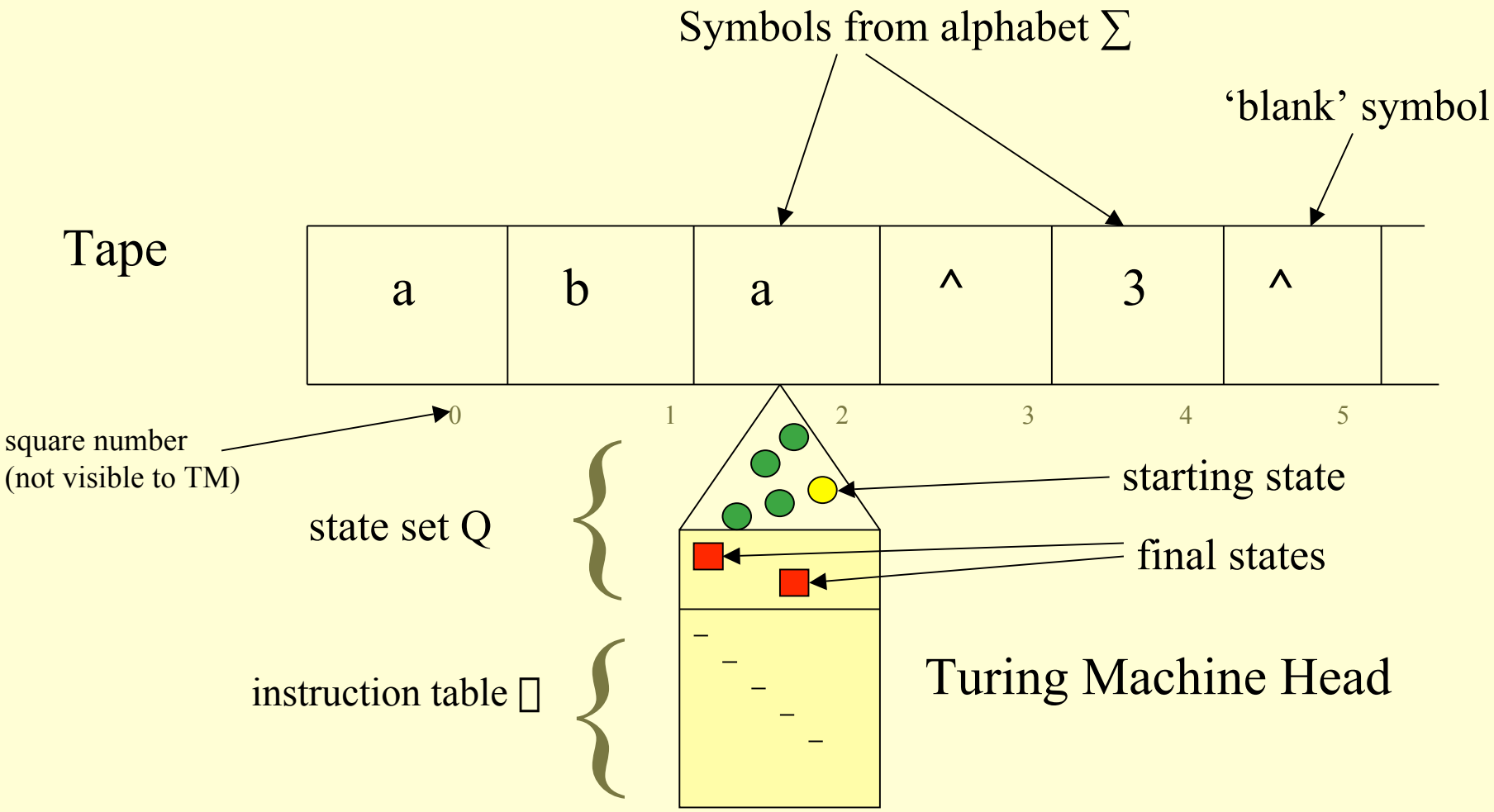
A Turing Machine



Turing Machines - a formal definition

A Turing Machine is a 6-tuple

$$M = (Q, \Sigma, I, q_0, \delta, F)$$

where

Q finite, non-empty set of states

Σ finite set of at least 2 symbols: the alphabet. $\wedge \delta \Sigma$

I non-empty subset of Σ; $\wedge \delta$ I; input alphabet

q_0 $q_0 \delta$ Q; starting or initial state

δ δ: (Q\F) x $\Sigma \delta$ Q x Σ x {-1, 0, 1}, a partial function, the instruction table

F F δ Q, the set of final or halting states

the Halting Problem

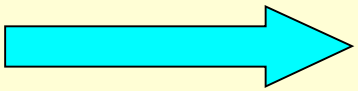
Will a given TM halt on a given input?

ie. Given as input:

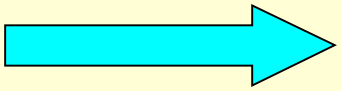
code(S) for a standard TM S
a word w of C

..can we determine whether S halts and succeeds?

We assume that we have a TM which determines whether S halts and succeeds - and derive a contradiction



Assumption is wrong



There is no such TM
The Halting Problem is unsolvable

Formal specification of the Halting Problem:

Define h: C* → C* such that

h(x) = 1 if x =code(S)*w for a standard TM, S, and
S halts and succeeds on input w.

= 0 if x=code(S)*w for some S, w, and
S does not Halt and Succeed
on input w

is undefined if x is not of the form code(S)*w for any
standard TM, S and input w.

h is a partial function C* → C*

is there a TM H such that $f_H = h$?

Such a TM would solve the Halting Problem.

Proof of the Halting Problem

assume Turing Machine H s.t. $f_H = h$

define a partial function g: g: C* → C* such that

$$g(w) = 1 \text{ if } h(w*w) = 0$$
$$\text{undefined otherwise}$$

Let M be a TM with $f_M = g$
M has a code, code(M)
[we know how to encode the alphabet if M is not standard, using only characters of C.]

Consider g(code(M))..it either has value 1 or is undefined..

1. Suppose g(code(M)) = 1
   - h(code(M)*code(M)) = 0 by defn. of g
   - M does not Halt and Succeed on input code(M) by defn. of h
   - f_M(code(M)) is undefined by defn. of Turing Machines
   - g(code(M)) is undefined…**CONTRADICTION**

2. Suppose g(code(M)) is not defined
   - f_M(code(M)) is not defined by defn of M
   - M does not Halt and Succeed on input code(M) by TM defn.
   - h(code(M)*code(M)) = 0 by defn of h
   - g(code(M)) = 1 …another **CONTRADICTION**

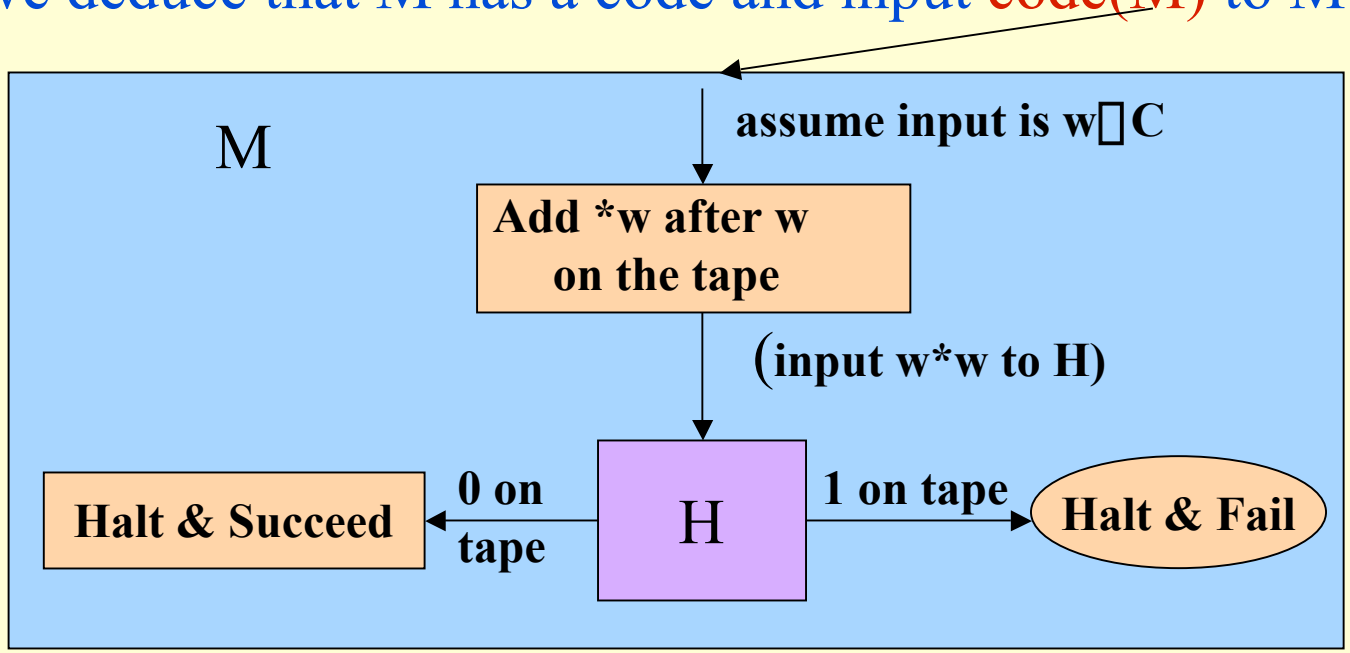
→ **erroneous assumption: H**

→ **there is no H** by the Church-Turing Thesis

the Halting problem is unsolvable

the Halting Problem diagrammatically..

we deduce that M has a code and input code(M) to M



M **Halts & Succeeds** iff output of H is 0
iff h(code(M)*code(M))=0
iff M **does not H & S** on input code(M)

M **Halts & Fails** iff output of H is 1
iff h(code(M)*code(M))=1
iff M **H & S** on input code(M).

CONTRADICTIONS

Consequences of Halting Problem unsolvability:

- we cannot write a program " to see whether our programs loop"

because this program (algorithm) would be implementable by a Turing Machine(by the Church-Turing thesis)
…and we have just shown that no such TM exists.

- we can use the Halting Problem result to prove **other** unsolvability results..

if we can show that a **solution to a new problem** could be used to build a **solution to the Halting Problem**..we know this is **impossible..**
..so we conclude that the **new problem must also be unsolvable.**

Summary

We have proved ..

..by assuming that the Halting Problem had a
Turing Machine (i.e. algorithmic) solution
and demonstrating that this leads to a contradiction,

that **no such TM exists and**

therefore the Halting Problem is unsolvable..

there is **no algorithmic solution**

Reduction:

We say that problem A **reduces** to another problem, B,
if we can convert any Turing Machine solution to B into a Turing machine solution to A
…if we can show how to adapt a solution to B to give a solution to A.

We might say that solving A is no harder than solving B

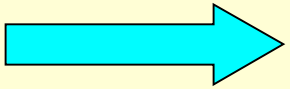
So if we know there is no TM solution to A, we deduce that there can be no TM solution to B either.

..important to get this argument the right way round..

Unsolvable problems

by Church's Thesis:

unsolvable by a Turing Machine



there is no algorithmic solution

This is independent of future hardware developments (eg faster machines)

Proof Methods

1. assume a solution exists..derive a contradiction..deduce the assumption was wrong

2. by reduction of a problem known to be unsolvable.

EIHP - the Empty Input Halting Problem

Does a TM Halt & Succeed on empty input □?

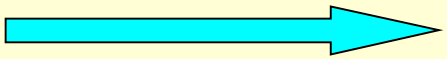
i.e. is there a TM EI such that for any standard TM S:

f_{EI}(code(S)) = 1 if S Halts & Succeeds on input □

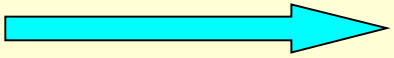
0 otherwise ?

This is proved by reducing HP to EIHP..

..we show that a solution to EIHP would provide a solution to HP..
known to be impossible..



EI cannot exist.

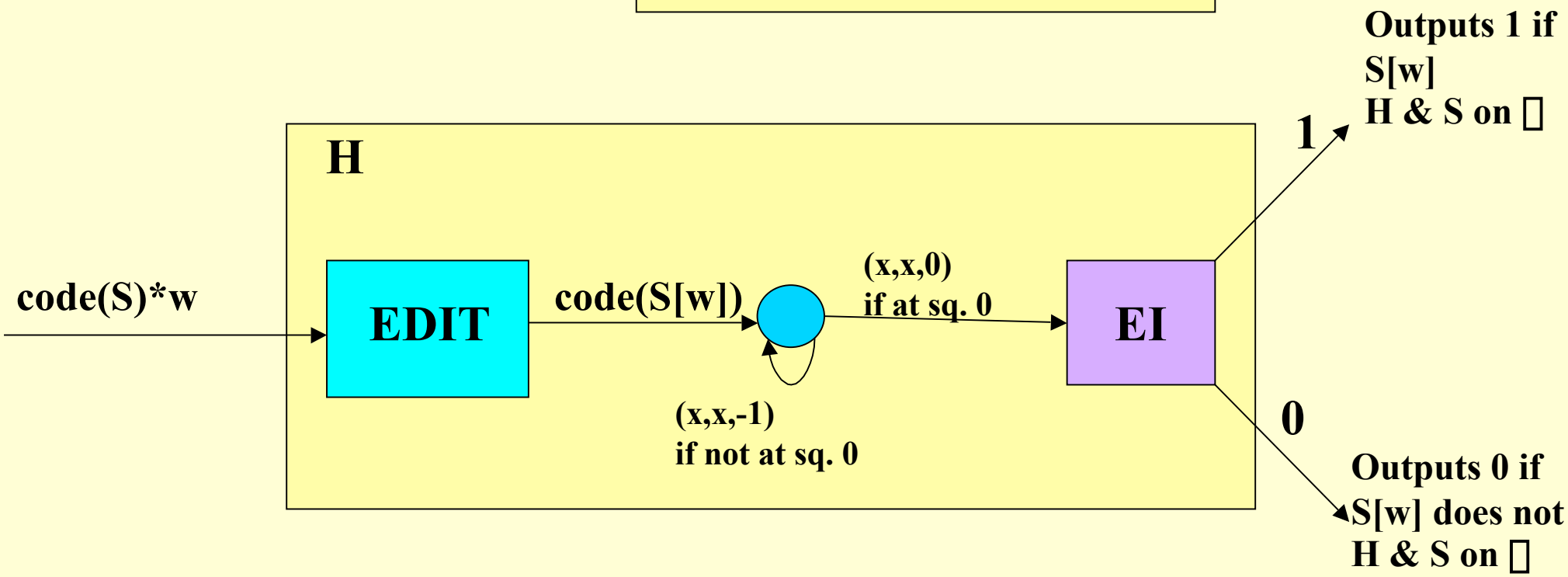


EIHP is unsolvable

1. Assume TM EI to solve EIHP

2. Define TM H by…

run EDIT
return to square 0
run EI



Let H have input code(S)*w.
H runs EDIT□ code(S[w]).

…which is input to EI which outputs
1 if S[w] Halts & Succeeds on input □
0 otherwise

S[w] Halts & Succeeds on input □
iff S Halts & Succeeds on input w

so: **H produces 1 on tape if S Halts & Succeeds on input w**
0 otherwise.

this is HP which has **NO** solution.
□ assumption that □ EI is false □ EIHP is unsolvable.

HP has been reduced to EIHP, proving EIHP unsolvable.

Summary..unsolvability results:

- we proved the Halting Problem unsolvable directly
  - by **assuming a solution** and **deriving a contradiction** from this assumption

- we proved EIHP (empty input halting problem) unsolvable
  - by **reducing the Halting Problem to EIHP**
  - showing that any solution to EIHP would provide a solution to the Halting Problem..
    …previous unsolvability result.

These methods can be used to prove many related results

P..the Class of tractable problems that can be solved efficiently (in polynomial time: p-time).

intractable problems are solvable but any algorithmic solution runs in exponential time (or slower) in the worst case. Practically unsolvable except for small inputs, unless average case much better than the worst.

NP..the class of problems which can be solved in p-time by a non-deterministic algorithm. Do they have deterministic p-time solutions? **"P = NP?"** if so, then all NP problems are in P..this has not been proved either way, but it is thought most likely that P ≠ NP, so problems in NP \P remain intractable (but not proved to be so).

NP-Complete problems..the hardest problems in NP. All NP-complete problems reduce to each other in p-time. Cook's theorem demonstrates that there are NP-complete problems (i.e. NP-complete is not an empty set)

Why do we study Complexity?..

- it guides us towards the tractable problems solvable with fast algorithms.
- ..but we often encounter NP-complete problems in practice..so it will avoid (practically) hopeless searches for fast algorithms.

- the reducibility of every NP-complete problem to every other gives us a higher level view of solvability and the notion of algorithm and its formalism by TMs.

We will:

- define the run time function of a Turing machine
- introduce non-deterministic TMs and their run-time function
- formalise fast reduction of one problem to another
- examine NP and NP-complete problems

The run-time function of a Turing Machine

$M = (Q, \sum, I, q_0, \square, F)$

for input words w of length n (n=1, 2, 3..):

M runs a varying number of steps for various words w of length n.

define

$time_M$ (n) = length of longest run of M for input of length n

the function

$time_M (n) : \{0, 1, 2, ..\} \rightarrow \{0, 1, 2, \ldots, \infty\}$

is the run-time function of M.

Tractable problems

A yes/no problem is **tractable** if it can be solved by a TM running in p-time

intractable if it can be solved algorithmically, but not in p-time

An algorithm is **tractable** if it can be implemented by a p-time TM

intractable if it cannot be implemented by a p-time TM.

(the Cook-Karp thesis: "p-time TMs are fast)).

P is the class of tractable problems :

i.e. they can be implemented by a p-time TM.

The **complement of a problem in P**: exchange yes and no

eg. is n prime? ⟷ (Complement) is n composite?

Summary

We have introduced:

the time function of a Turing Machine
polynomial time function (p-time) TMs

Tractable and Intractable problems and algorithms

Complexity classes of problems

P ..can be solved by a deterministic TM in p-time

(for NP and NP-complete see later lectures).

Formal definition of NDTM

$N = (Q, \sum, I, q_0, \square, F)$, with

$\square : (Q \backslash F) \times \sum \square \; 2^{Q \times \sum \times \{\pm 1, 0\}}$

- $: 2^{Q \times \sum \times \{\pm 1, 0\}}$: the set of all subsets of $Q \times \sum \times \{\pm 1, 0\}$

ie. the function value for (q,a) is a set of the alternatives

If there is no applicable instruction for (q,a) then $\square(q,a) = \varnothing$ (empty set).

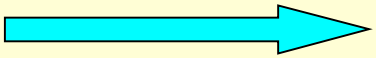
For a TM, M:

if $\square(q,a)$ contains only one (q',a',d) or is empty, we have an ordinary deterministic TM.

Polynomial -time Reduction

We formalise reduction by
defining p-time reduction in terms of Turing Machines.

fast non-deterministic solutions to old yes/no problems



fast non-deterministic solutions to new ones.

Definition of p-time reduction ' ≤ '

let A, B be any two yes/no problems
X a deterministic Turing Machine

X reduces A to B if: for every yes-instance w of A, $f_X(w)$ is defined
and is a yes-instance of B
for every no-instance w of A, $f_X(w)$ is defined
and is a no-instance of B

A reduces to B in p-time if ∃ a det TM X running in p-time that reduces A to B (**A ≤ B** if A reduces to B in polynomial time).

If A ≤ B and B ≤ A ⇒ A ~ B.

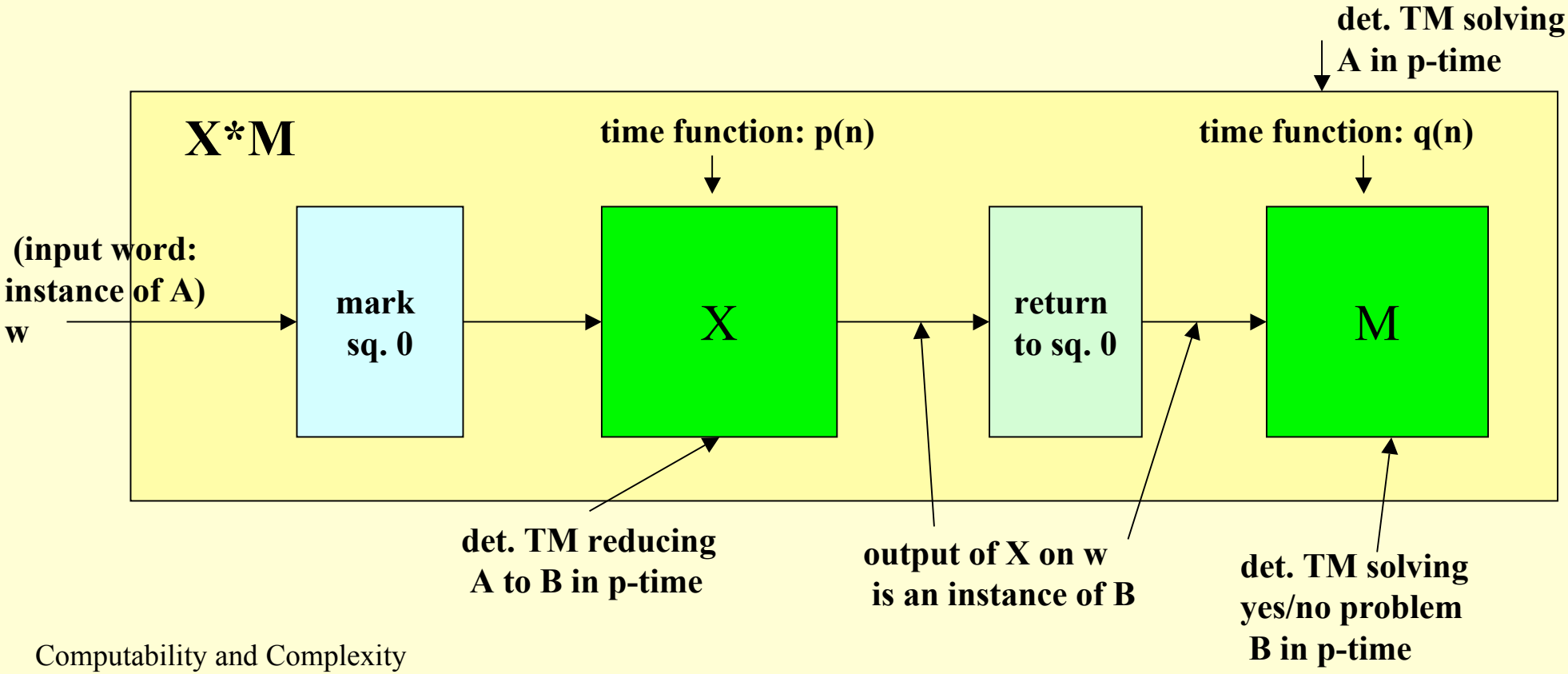
P is closed downwards under reduction

if $A \leq B$ and $B \in P$..then $A \in P$.

time function of X*M: $\text{time}_{X*M}(n) \leq 1 + p(n) + p(n) + q(p(n))$

a polynomial

X*M is a deterministic p-time TM which solves A



The Class NP of problems

NP consists of all yes/no problems A such that there is some **NDTM N that runs in p-time and solves A;**

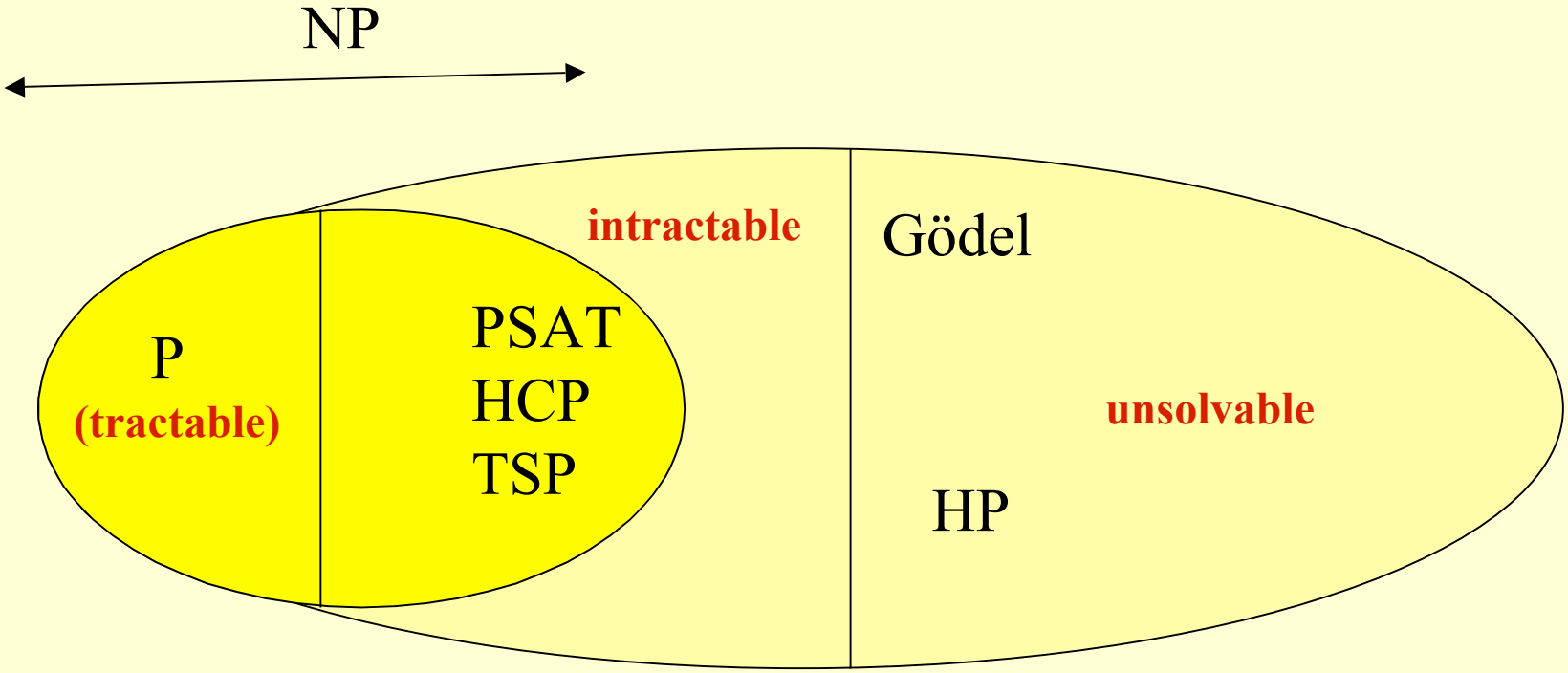
N accepts all the yes-instances of A
rejects all the no-instances of A.

This is the class of (□) type problems that would be in P if they had a clever search strategy.

P ⊆ NP as p-time deterministic TMs are a special case of p-time non-deterministic TMs.

P = NP? .. yes/no problem not yet answered

≤ yes/no problems



Complexity classes P, NP.

NP-complete problems..NPC

is there a ≤-hardest problem in NP. Or set of hardest problems?

or..a sequence ≤ harder ≤ harder ≤ harder ≤…

there are hardest problems in NP: the NP-complete problems.

Definition of NP-complete:

A yes/no problem A is NP-complete if:

1. A □ NP
2. B ≤ A for all problems B □ NP

   i.e. NP-complete problems are problems in NP to which all other NP Problems can be reduced in p-time.

NPC - the class of NP-complete problems.

If A, B are NP-complete then A~B:

A is NP-complete ⇔ A ∈ NP

for all C ∈ NP, C≤A (including B)

B is NP-complete ⇔ B ∈ NP

for all C ∈ NP, C≤B (including A)

..so A≤B and B≤A .. **A~B**.

If A is NP-complete and A~B then B is NP-complete:

A is NP-complete ⇔ A ∈ NP

⇔ for all C ∈ NP, C≤A

A~B ⇔ A≤B and B≤A.

NP is closed downwards under ≤ ⇔ B ∈ NP

For any C ∈ NP, C≤A and A≤B so as ≤ is transitive, C≤B.

so **B is NP-complete**.

A yes/no problem A is NP-complete if:

1. A ∈ NP
2. B ≤ A for all problems B ∈ NP

if we know that another problem C, is NP-complete, we can show

2*. C ≤ A (instead of 2. above)

(remember 1. must be shown: that A ∈ NP.

This permits extension of the set NPC by proving that a known NP-complete problem reduces in p-time to an NP problem thought to be in NPC…but is there a "first" problem in NPC?

to use 1.+and 2*) as proof, we need an existing NP-complete problem

are there any NP-complete problems?

Cook's Theorem proved that PSAT is NP-complete

so NPC ≠ ∅