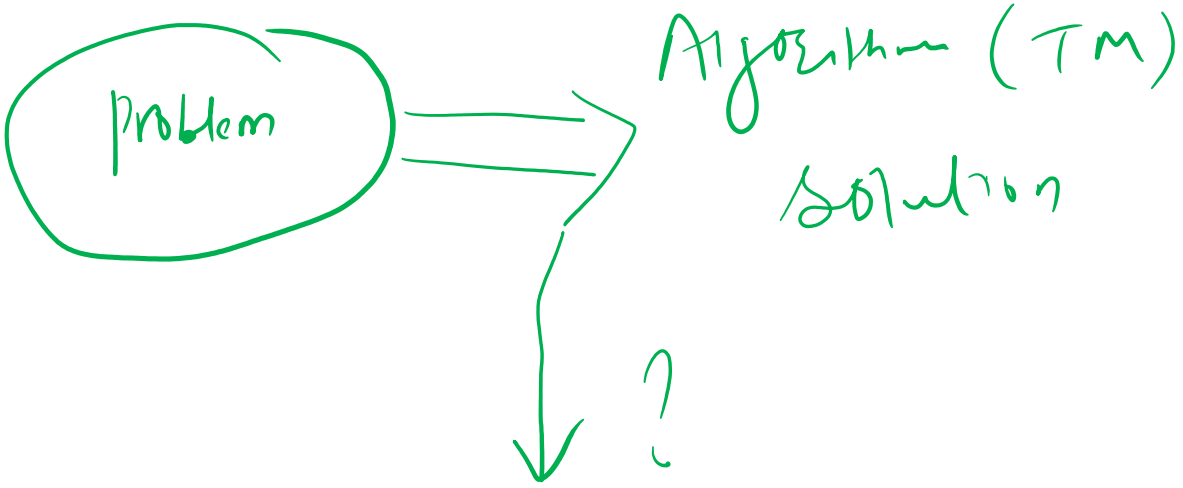
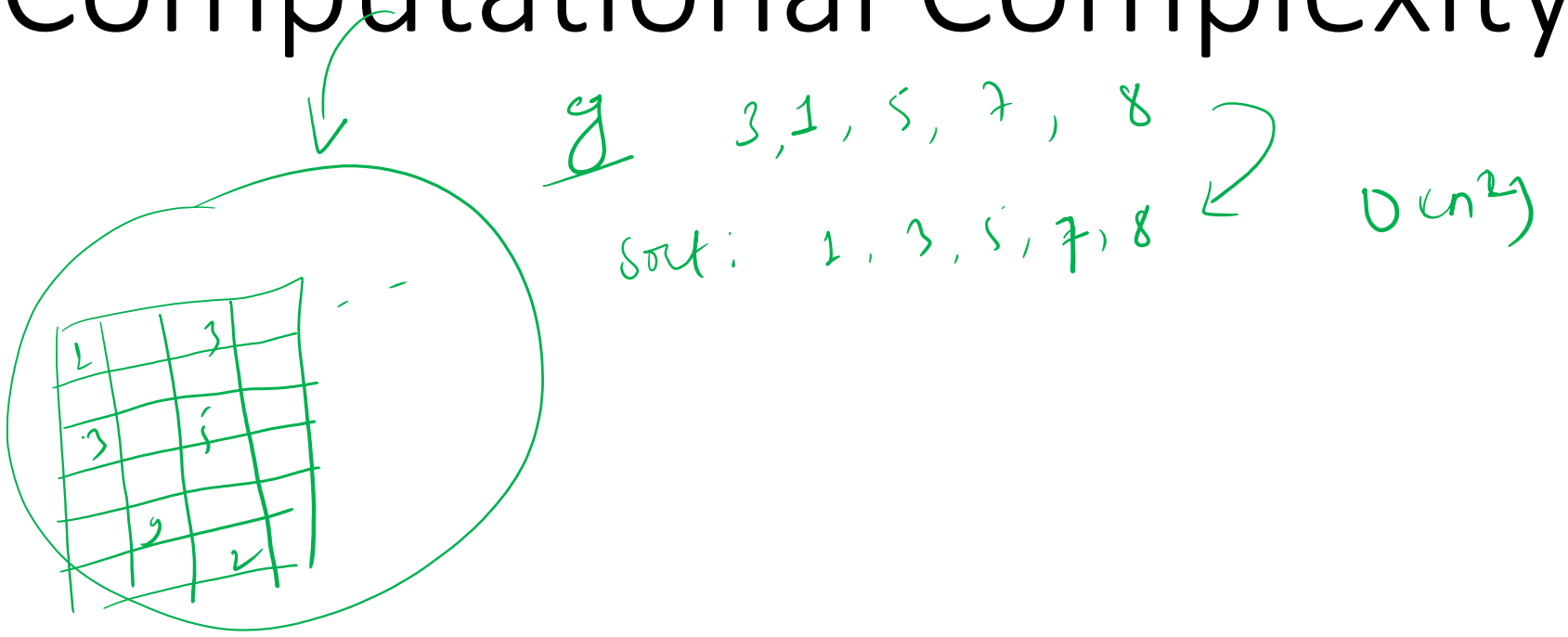
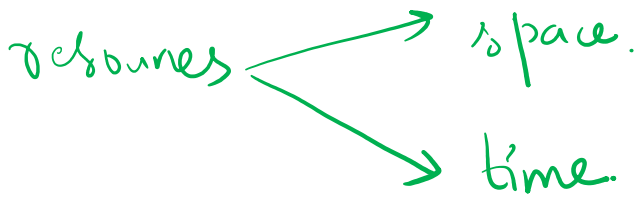


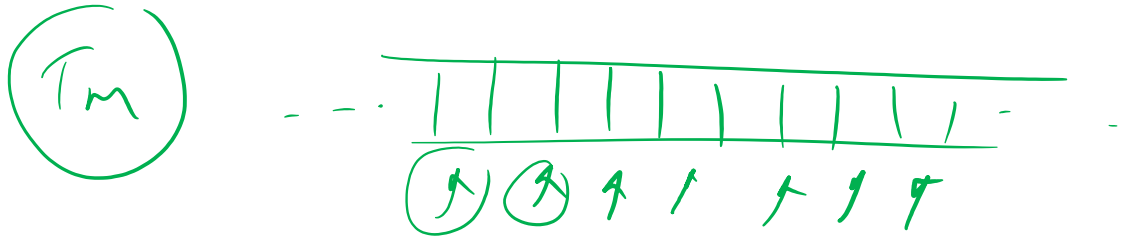
Computational Complexity





- Complexity theoretic study looks at a task (or a class of tasks) and at the computational resources required to solve this task, rather than at a specific algorithm or algorithmic scheme.

- The complexity of computational problems can be discussed by choosing a specific abstract machine as a model of computation and considering how much resource machine of that type require for the solution of that problem.



space / time

- Complexity Measure is a means of measuring the resource used during a computation.
- In case of Turing Machines, during any computation, various resources will be used, such as space and time.

Time Complexity

The most critical computational resource is often time, so the most useful complexity measure is often time complexity

If we take Turing Machine as our model of computation, then we can give a precise measure of the time resources used by a computation

Definition The time complexity of a Turing Machine T is the function Time_T such that $\text{Time}_T(x)$ is the number of steps taken by the computation $T(x)$

(Note that if $T(x)$ does not halt, then $\text{Time}_T(x)$ is undefined.)

Space Complexity

Another important computational resource is amount of "memory" used by an algorithm, that is space. The corresponding complexity measure is space complexity

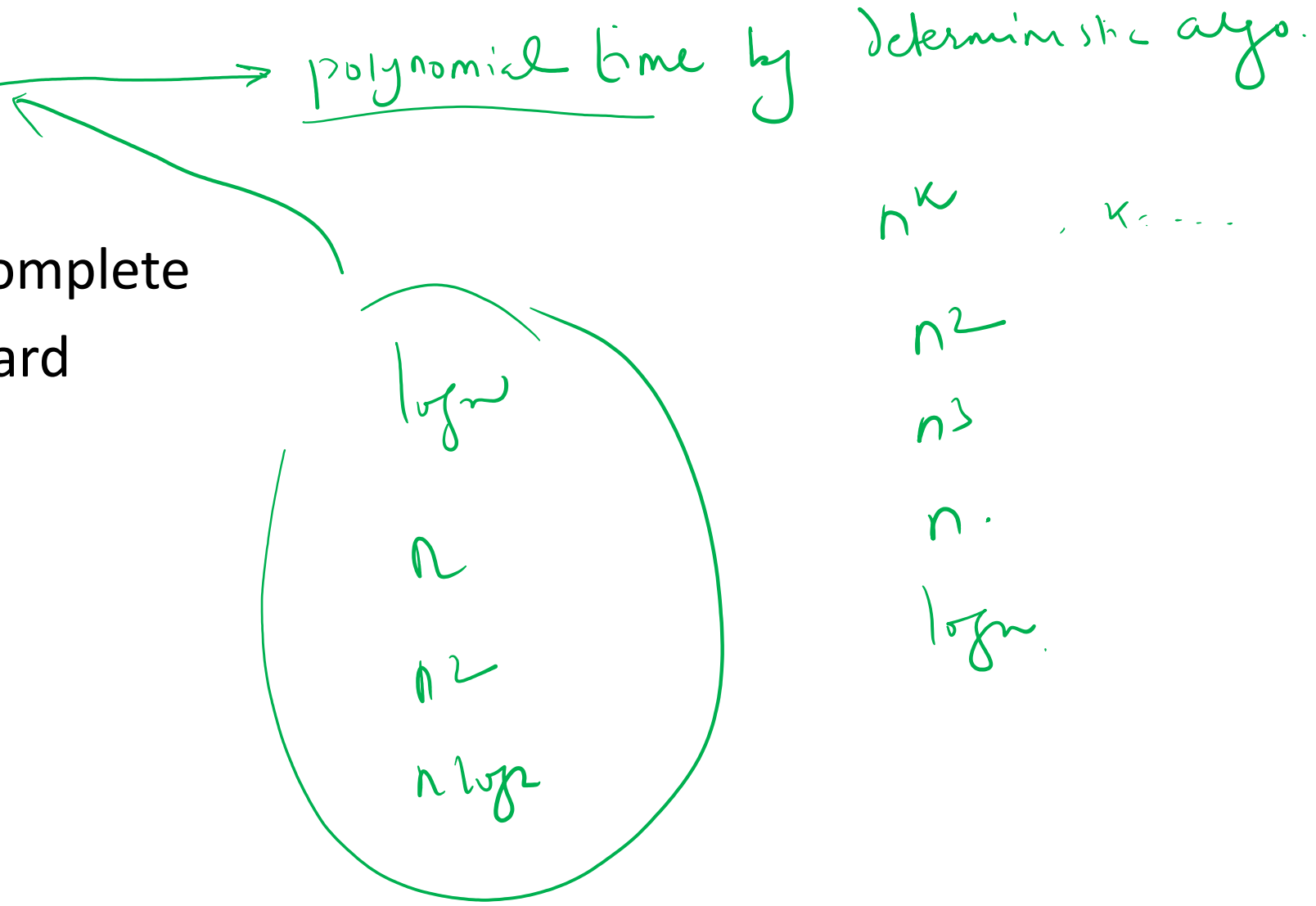
As with time, if we take Turing Machine as our model of computation, then we can easily give a measure of the space resources used by a computation

Definition The space complexity of a Turing Machine T is the function Space_T such that $\text{Space}_T(x)$ is the number of distinct tape cells visited during the computation $T(x)$

(Note that if $T(x)$ does not halt, then $\text{Space}_T(x)$ is undefined.)

Complexity Classes

- P
- NP
- NP Complete
- NP Hard



The class P

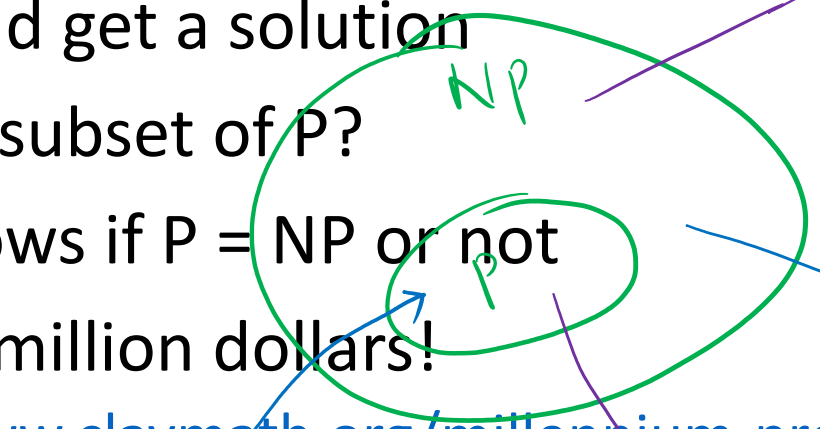
- The class P consists of those problems that are solvable in polynomial time.
- More specifically, they are problems that can be solved in time $O(n^k)$ for some constant k, where n is the size of the input to the problem
- The key is that n is the **size of input**

NP (XX → non-polynomial) XX

- **NP is not the same as non-polynomial complexity/running time. NP does not stand for not polynomial.**
- **NP = Non-Deterministic polynomial time**
- NP means verifiable in polynomial time
- Verifiable?
  - If we are somehow given a 'certificate' of a solution we can verify the legitimacy in polynomial time

P is a subset of NP

- Since it takes polynomial time to run the program, just run the program and get a solution
- But is NP a subset of P?
- No one knows if P = NP or not
- Solve for a million dollars!
  - http://www.claymath.org/millennium-problems
  - The Poincare conjecture is solved today



P vs NP Problem

- If it is easy to check that a solution to a problem is correct, is it also easy to solve the problem?
- This is the essence of the P vs NP question.
- Typical of the NP problems is that of the Hamiltonian Path Problem: given N cities to visit, how can one do this without visiting a city twice?
- If you give me a solution, I can easily check that it is correct. But I cannot so easily find a solution.

NP

Solve by non-deterministic algo in polynomial time

Verification in polynomial time

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

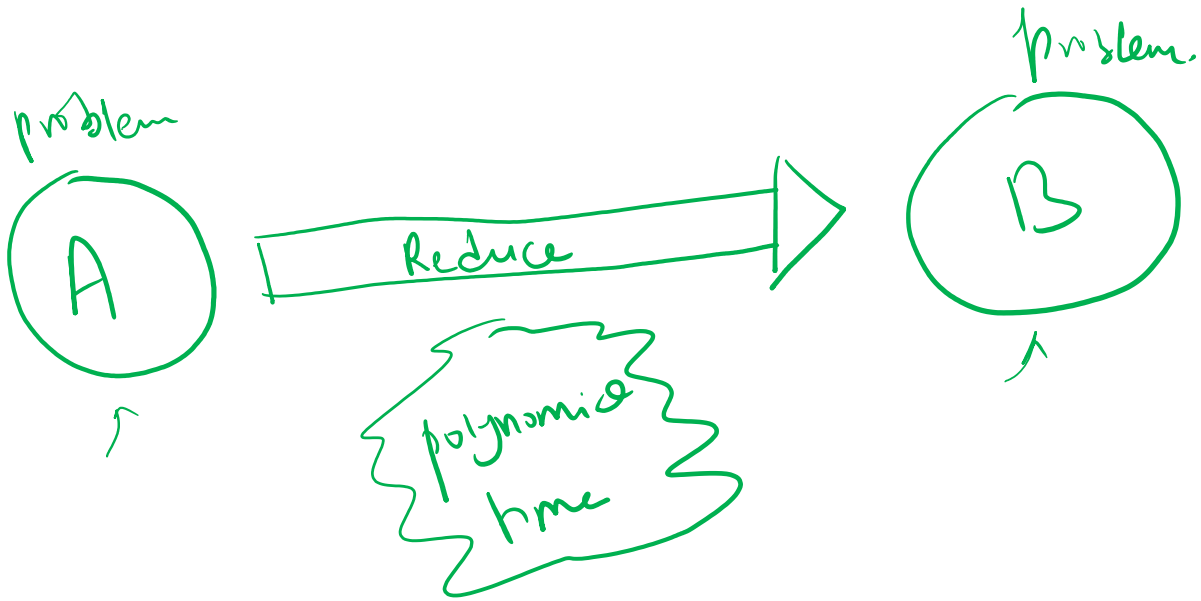
Is $P = NP$?

$\Rightarrow$ Online security is vulnerable to attack

Is $P \neq NP$?

$\Rightarrow$ There are some problems that can never be solved

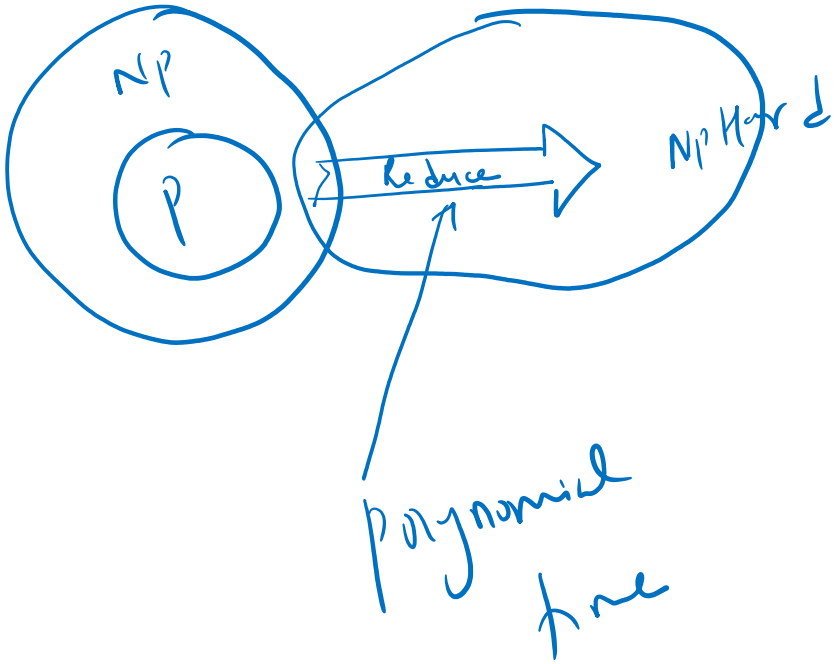
Reduction :



problem A reduces to problem B iff there is a way to solve A by deterministic algorithm that solve B in polynomial time.

Np-Hard problem:

A problem is Np-Hard if every problem in NP can be polynomial reduced to it.



Np- Complete

1) Np- Complete problem :
Decision problem (Y/N)

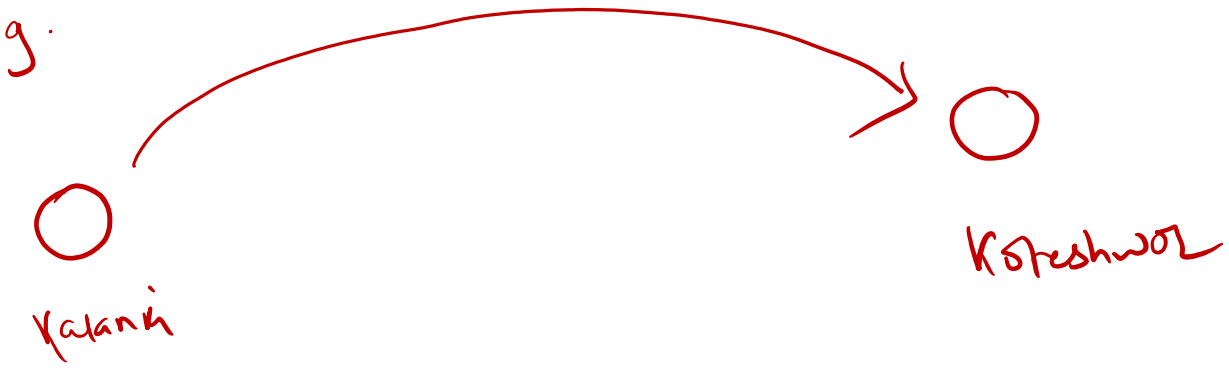
2) Np- Hard problem :
optimization problem (Minimum/Maximum)

3) All Np- Complete problems are Np- Hard but all Np- Hard problem are not Np- complete

1) Decision problem:

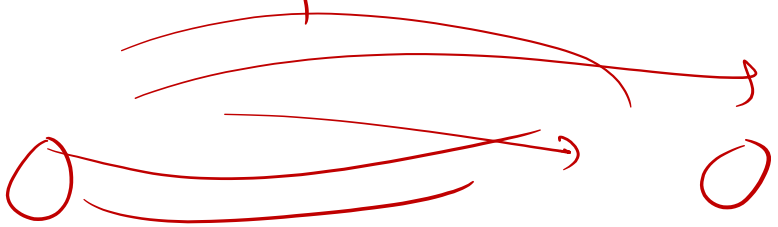
is there shortest path to reach to Kateshwor ! → Yes / No

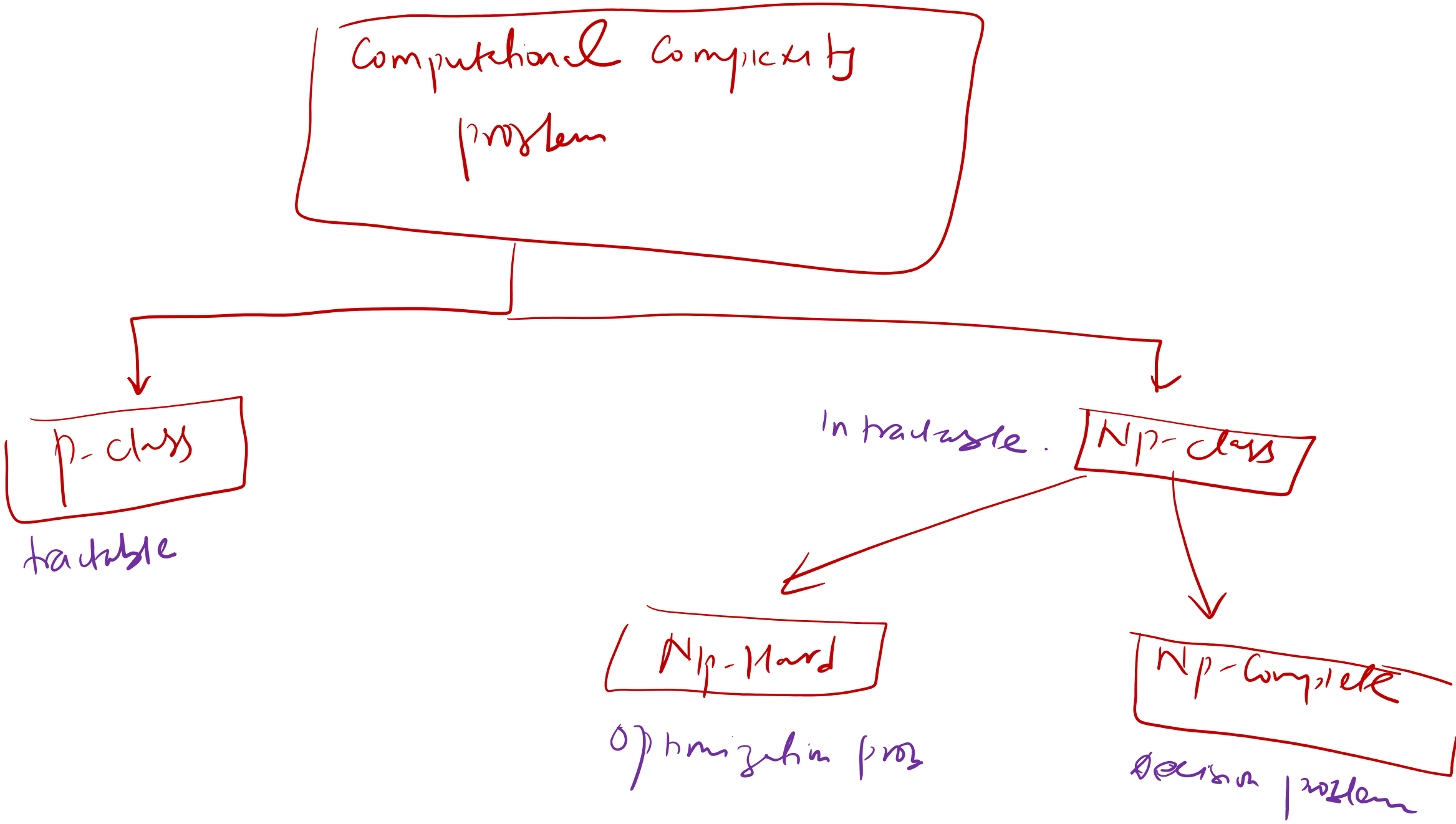
eg.



2) Optimization problem.

find out the best route.




Computational Complexity
problem
P - class
tractable
Intractable.
NP - class
NP - Hard
Optimization prob
NP - Complete
Decision problem

Reducibility

- Reducibility is a way of converging one problem into another in such a way that, a solution to the second problem can be used to solve the fist one.
- For instance, if a problem X can be solved using an algorithm for Y, X is no more difficult than Y, and we say that X reduces to Y.
- The most commonly used reduction is a polynomial-time reduction.
- For example, the problem of squaring an integer can be reduced to the problem of multiplying two integers.
- This means an algorithm for multiplying two integers can be used to square an integer.

- Indeed, this can be done by giving the same input to both inputs of the multiplication algorithm.
- Thus we see that squaring is not more difficult than multiplication, since squaring can be reduced to multiplication.

SAT Problem / Satisfiability

eg.1 $F = (a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee c) \wedge$

$(\neg c \vee b) = 1$

$a = 1, b = 1, c = 1, F = 1$

eg2. $F = (\overset{1}{a} \vee \overset{1}{b}) \wedge (\overset{1}{a} \vee \neg \overset{0}{b}) \wedge (\neg \overset{0}{a} \vee \overset{1}{c}) \wedge$

$(\neg \overset{0}{c} \vee \neg \overset{0}{a}) = 0$

Given:

Boolean exp^n $(\wedge, \vee, \neg)$

Assign Boolean value to operand $(0, 1)$

Check: False / True of entire expression

e.g. $F = 1$ (satisfiable)

$F = 0$ (not satisfiable)

Boolean SAT ⟶ SAT

e.g. $\neg(p \wedge q)$

p	q	$\neg(p \wedge q)$	
0	0	1	satisfy.
1	1	0	
1	0	1	
0	1	1	

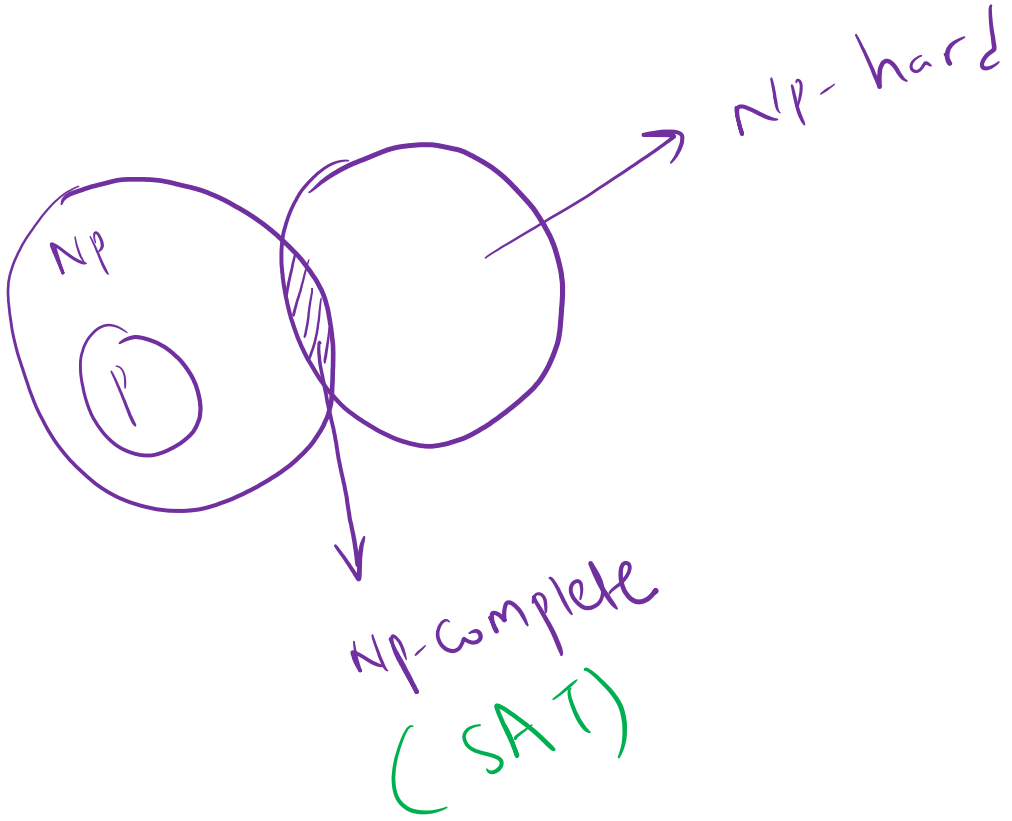
$p \wedge \neg p$

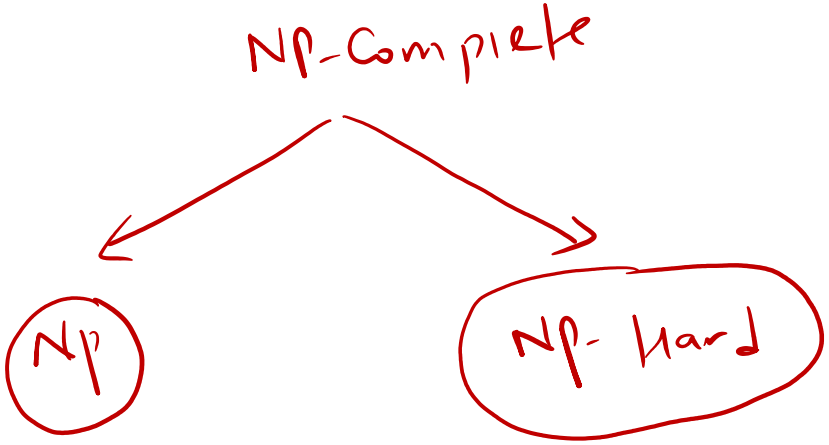
p	$\neg p$	$p \wedge \neg p$	
0	1	0	
1	0	0	

not satisfiable

Cook's Theorem :

Stmt: SAT is NP-Complete problem.





How SAT is NP ?

- Solve by NDA/NDTM in PT. (exponential)
- Verifiable in PT.

eg. $F = (a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee c) \wedge (\neg c \vee b)$

a	b	c
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

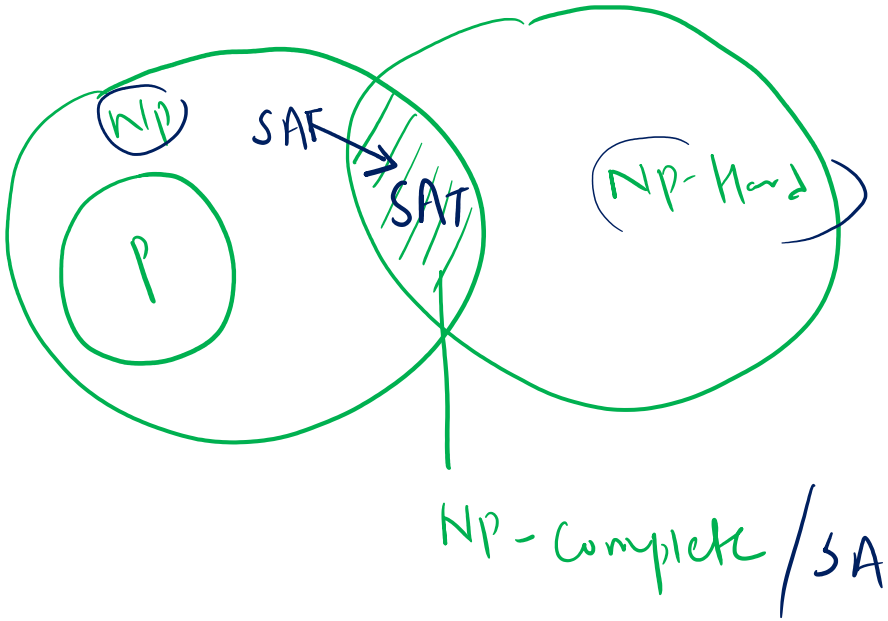
$2^3 = 8$

for n variable

$2^n \longrightarrow$ exponential

How SAT is NP-Hard ?

→ reduce in polynomial time



NP-complete / SAT

Continue.

Reduction in NP-Complete:

If A and B are two lang. And if A is reducible to B in polynomial time then B is NP-Hard.

If B is in NP also, then B is NP-Complete.

Steps for proving NP-complete:

1) prove that B is in NP.
2) Select an NP-complete language A.
3) Construct a function f that maps members of A to member of B.
4) Show that x is in A iff f(x) is in B.
5) show that f can be computed in polynomial time.

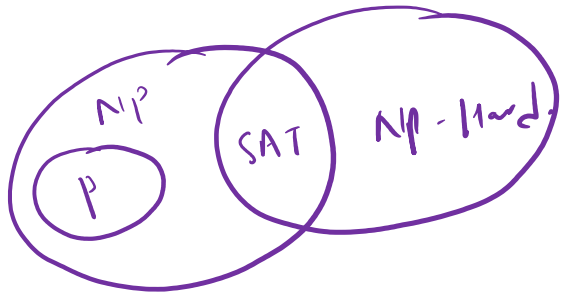
Cook's Theorem:

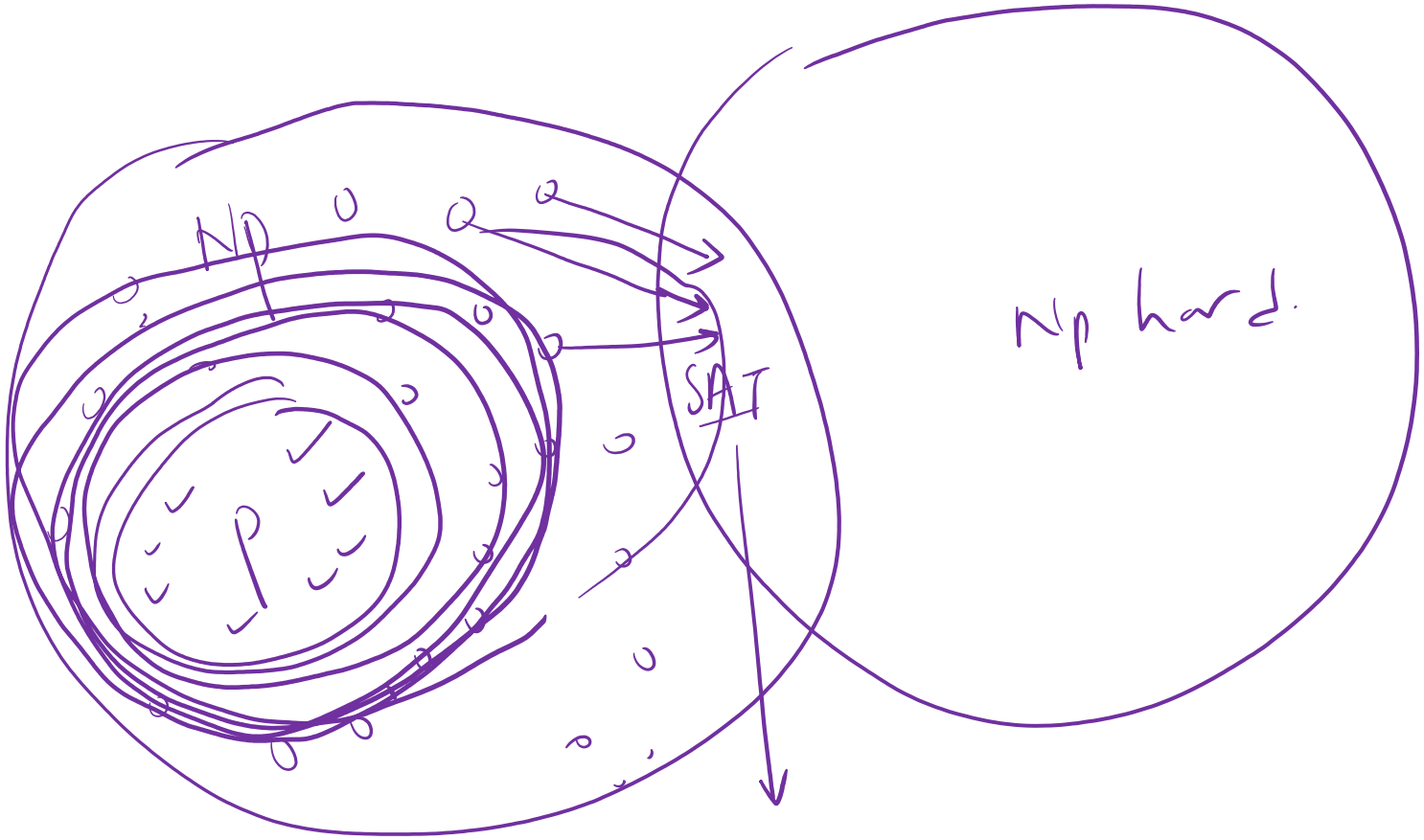
↳ SAT is NP Complete.

Supporting statements.

↳ SAT is in P iff P = NP.

↳ P ≠ NP is SAT belongs to P.





1st Np. Complete problem

P → class of problems that can be solved by D.A in PT

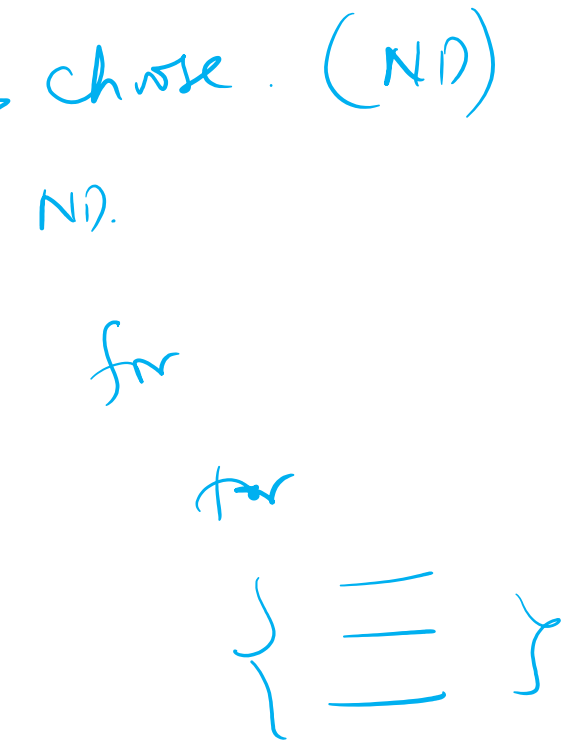
(sorting, searching) ...)

NP → solve in exponential time / verify PT

TSP, VC

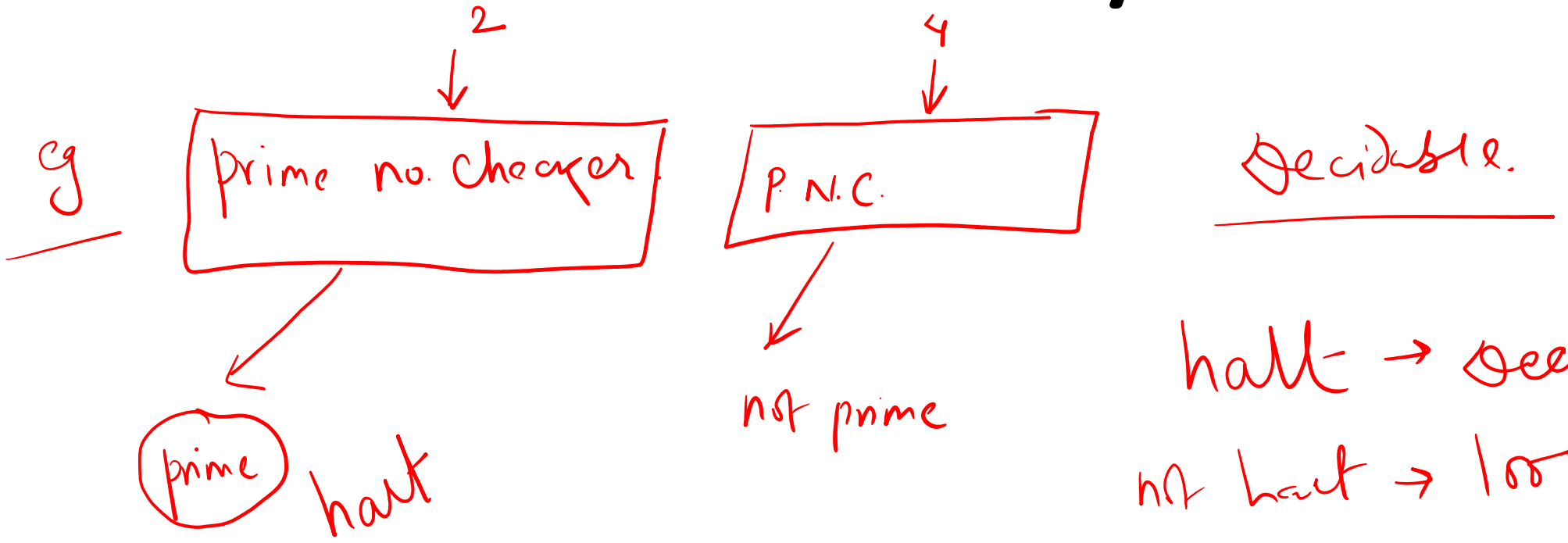
Nondeterministic sorting

```
B ← 0
/* guessing */
for i = 1 to n do
    j ← choice(1 : n)
    if B[j] ≠ 0 then failure
    B[j] = A[i]
/* checking */
for i = 1 to n-1 do
    if B[i] > B[i+1] then failure
success
```



P.N.C. Algo/program/TM.

Undecidability



Decidability vs. Undecidability

- There are two types of TMs (based on halting):

(*Recursive*)

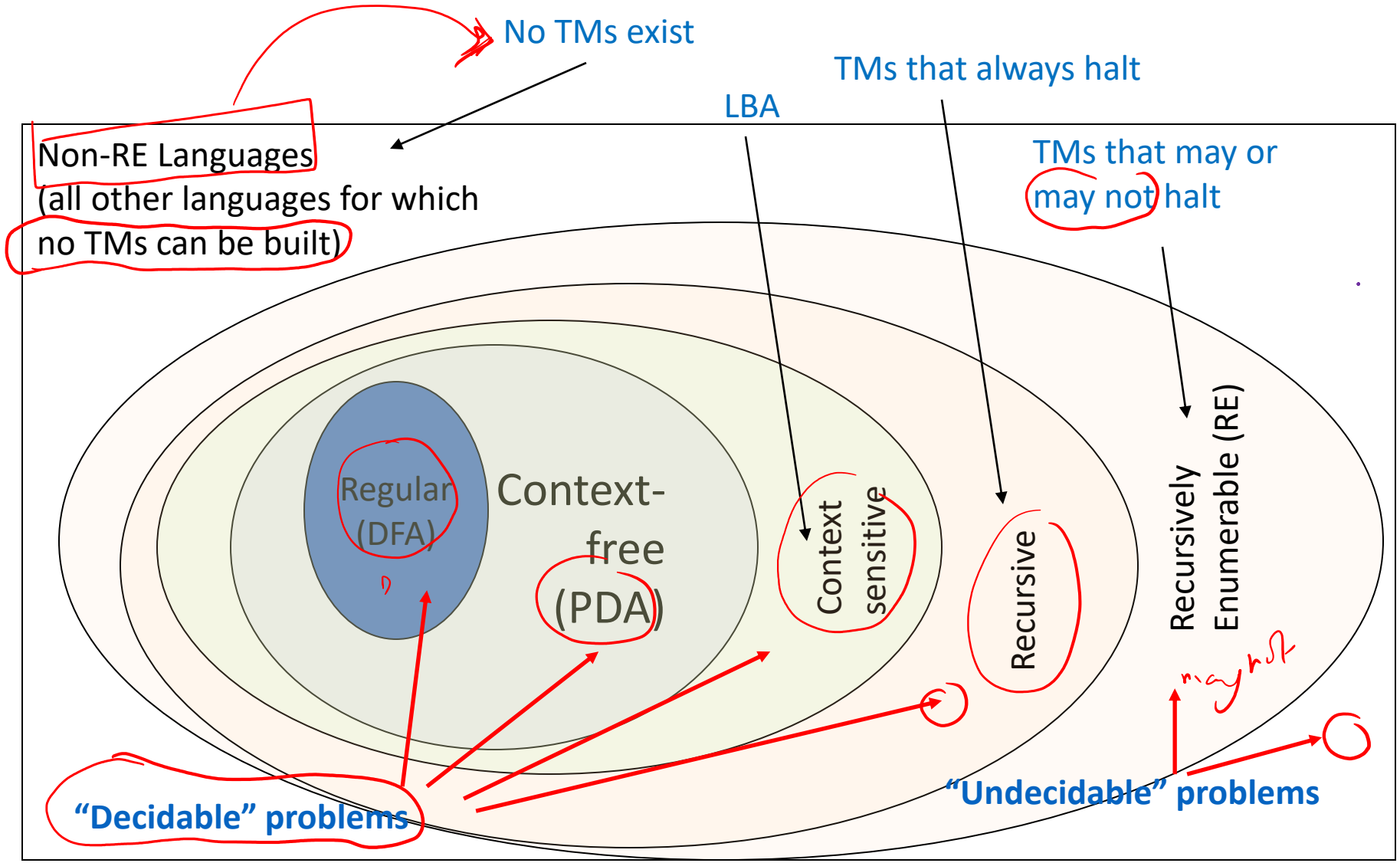
TMs that *always* halt, no matter accepting or non-accepting ≡ **DECIDABLE** PROBLEMS

(*Recursively enumerable*)

TMs that *are guaranteed to halt* only on acceptance. If non-accepting, it may or may not halt (i.e., could loop forever).

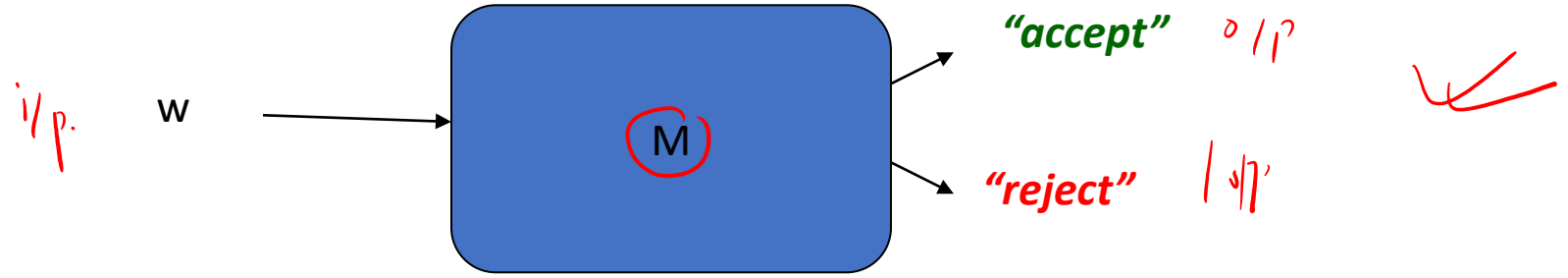
- **Undecidability:**
  - Undecidable problems are those that are not recursive

Recursive, RE, Undecidable languages

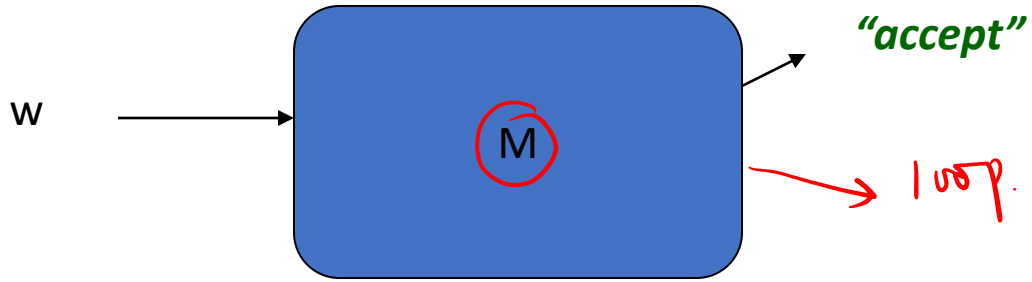


Recursive Languages & Recursively Enumerable (RE) languages

- Any TM for a Recursive language is going to look like this:



- Any TM for a Recursively Enumerable (RE) language is going to look like this:

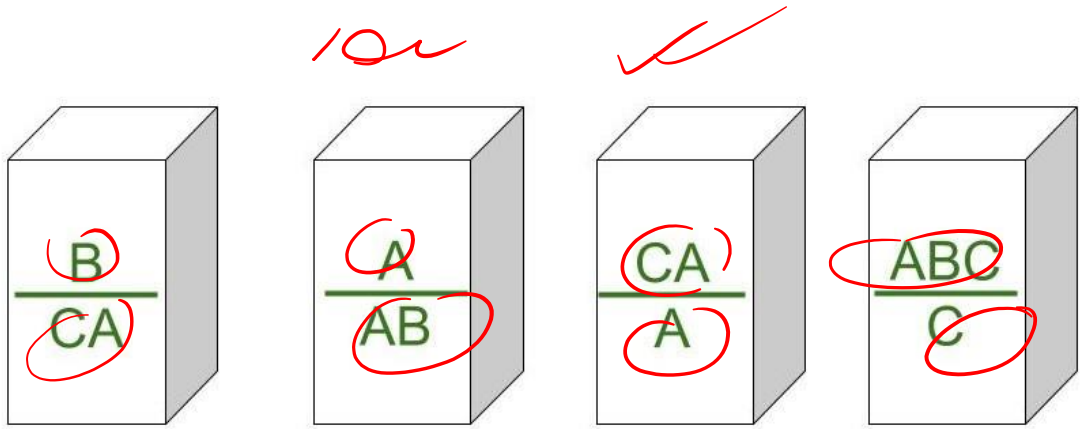


Undecidable Problems

1. **Post Correspondence Problem (PCP)**
   - **Illustration**
2. **Halting Problem** (Question rather than a problem)
   - **Illustration**
   - **Proof**

1. Post Correspondence Problem:

- Post Correspondence Problem is a popular undecidable problem that was introduced by **Emil Leon Post** in **1946**.
- In this problem we have N number of Dominos (tiles). The aim is to arrange tiles in such order that string made by Numerators is same as string made by Denominators.

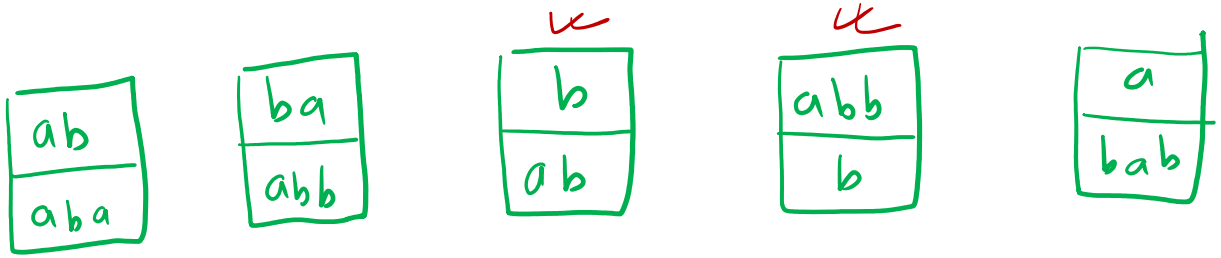


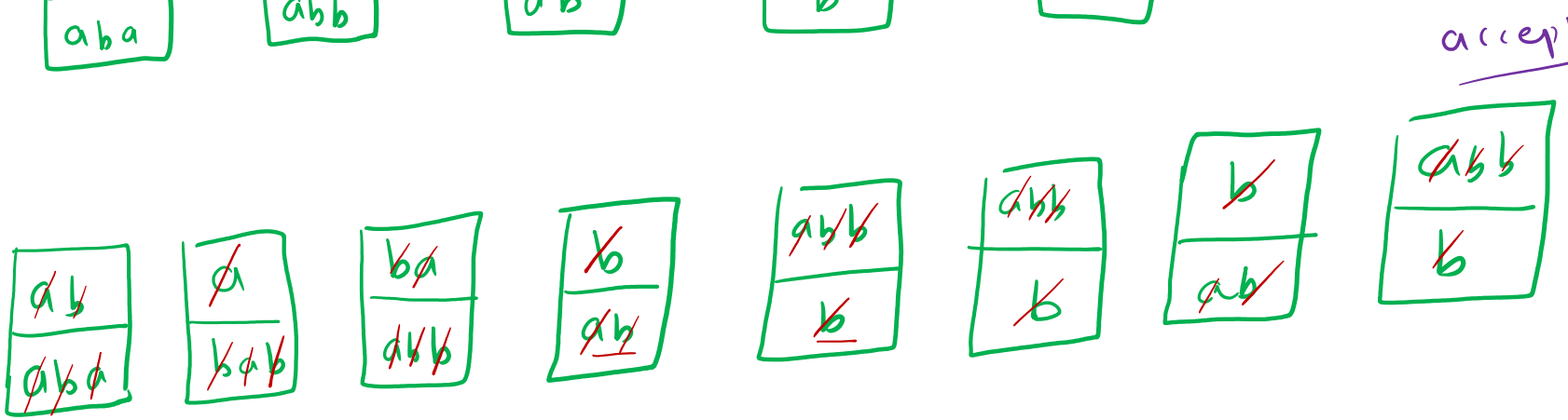
	Numerator	Denominator
1	B	CA
2	A	AB
3	CA	A
4	ABC	C

b, ba, bab

a, abb, ab

Illustration





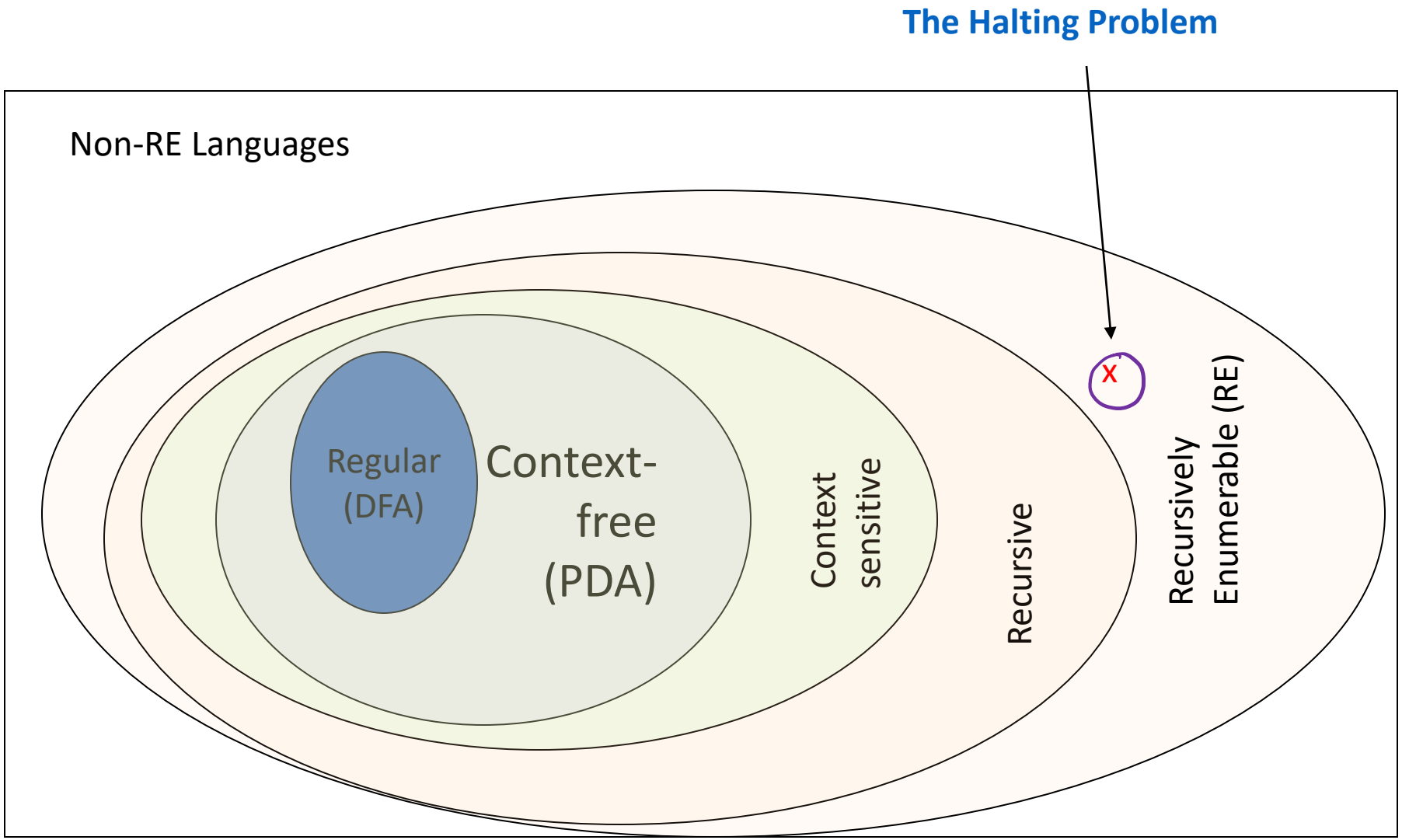
aba bab abbabb babb

aba ba babbab bbabb

The Halting Problem

An example of a recursive enumerable problem that is also undecidable

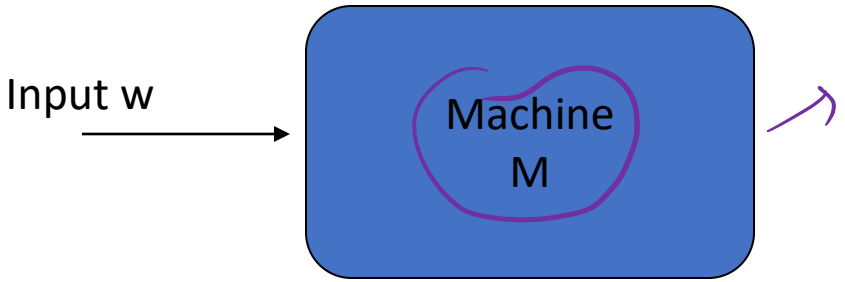
may / may not.


The Halting Problem
Non-RE Languages
Regular (DFA)
Context-free (PDA)
Context sensitive
Recursive
Recursively Enumerable (RE)
X

What is the Halting Problem?

Definition of the "halting problem":

- *Does a given Turing Machine M halt on a given input w?*



- Halting Problem Proof Animation

A Turing Machine simulator

The Universal Turing Machine

- Given: TM **M** & its input **w**
- Aim: Build another TM called "H", that will output:
  - "*accept*" if M accepts w, and
  - "reject" otherwise

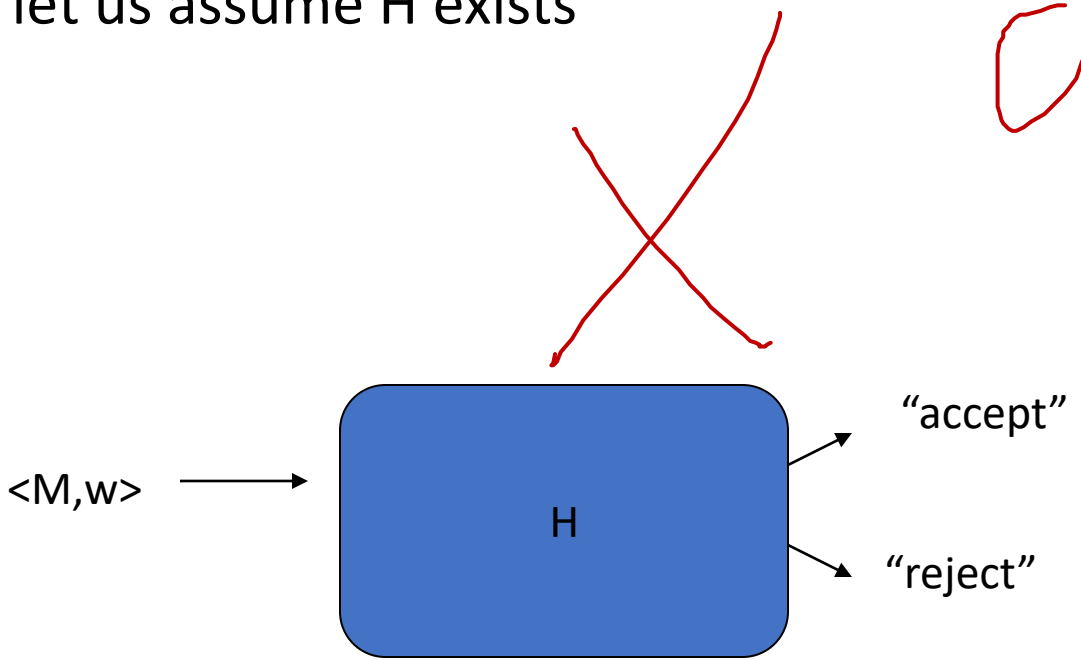
Implies: H is in RE

- An algorithm for H:
  - Simulate M on w
  - H(<M,w>) = { *accept*, if *M* accepts w ; *reject*, if M does does not accept w }

Question: If M does *not* halt on w, what will happen to H?

A Claim

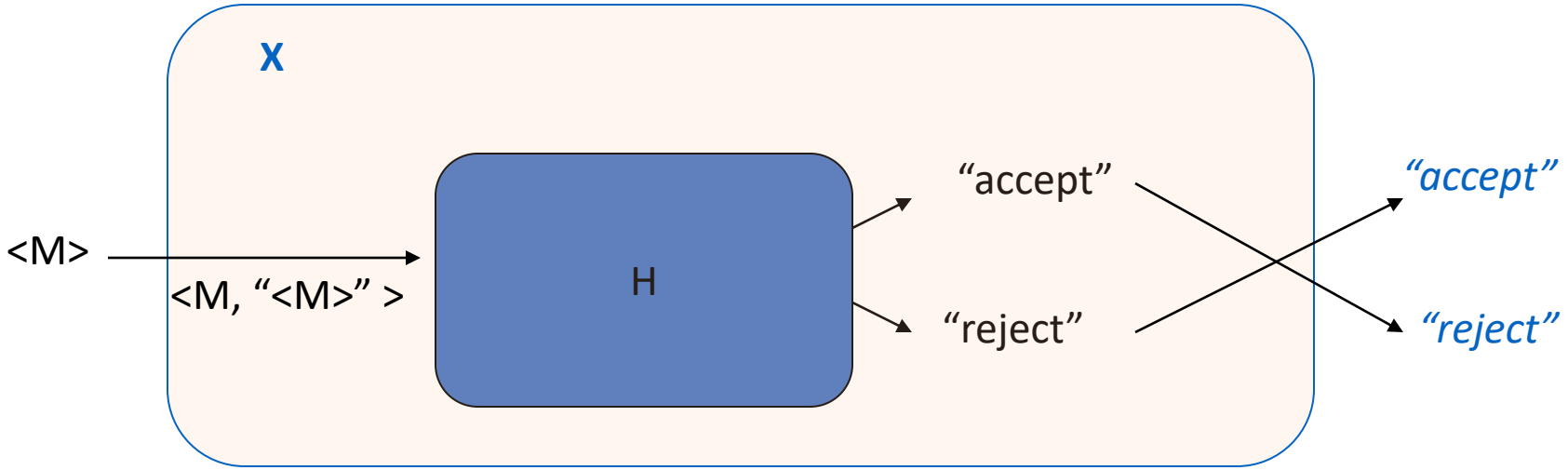
- Claim: No H that is always guaranteed to halt, can exist!
- Proof: (Alan Turing, 1936)
  - By contradiction, let us assume H exists



Therefore, if H exists ➔ X also should exist.
But can such a X exist? (if not, then H also cannot exist)

HP Proof (step 1)

- Let us construct a new TM **X** using H as a subroutine:
  - On input <M>:
    1. Run H on input <M, <M> >; //(i.e., run M on M itself)
    2. Output the *opposite* of what H outputs;



HP Proof (step 2)

- The notion of inputing "<M>" to M itself
  - A program can be input to itself (e.g., a compiler is a program that takes any program as input)

X (<M>) =
- *accept*, if M does *not* accept <M>
- *reject*, if M accepts <M>

Now, what happens if X is input to itself?

X (<X>) =
- *accept*, if X does not accept <X>
- *reject*, if X accepts <X>

A contradiction!!! ==> Neither X nor H can exist.