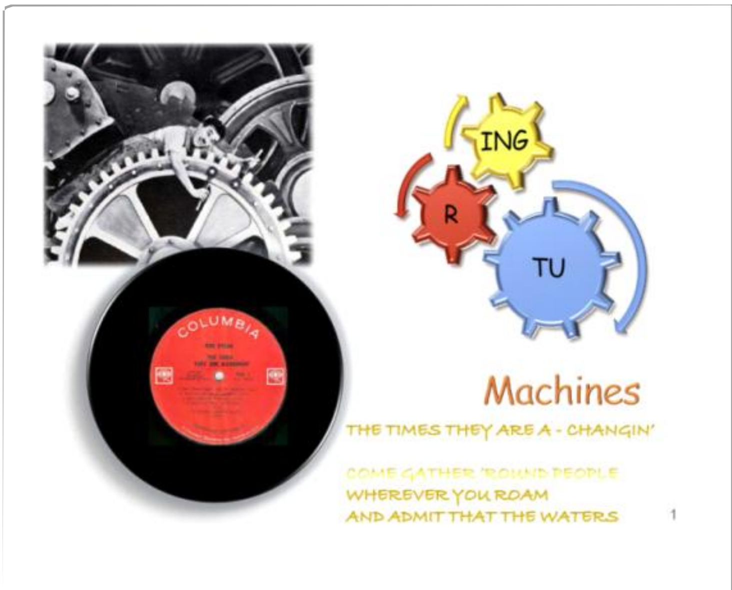
TM notes



The purpose here is to classify computational problems according to their complexity.

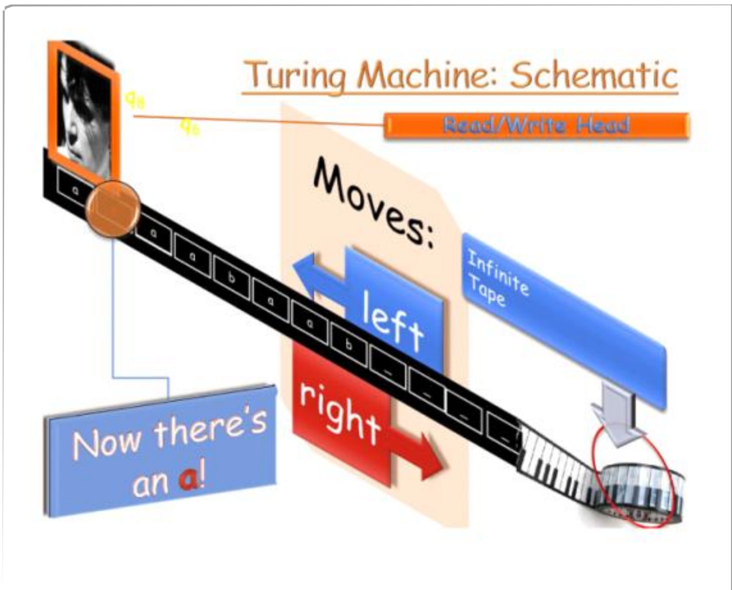
For that purpose, we need first to agree on a computational model.

We'll remind you what a Turing Machine is --- you did study about it in previous courses.

This time we will introduce some bounds by which one can introduce some complexity classes.

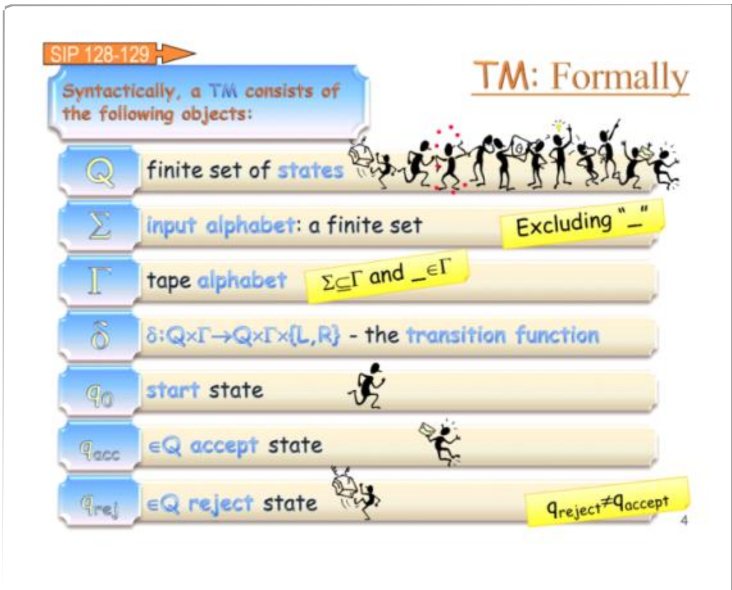
We'll go over different types of Turing machines.

Turing Machines
Turing Machines (2)
Turing Machines (3)
Church-Turing Hypothesis



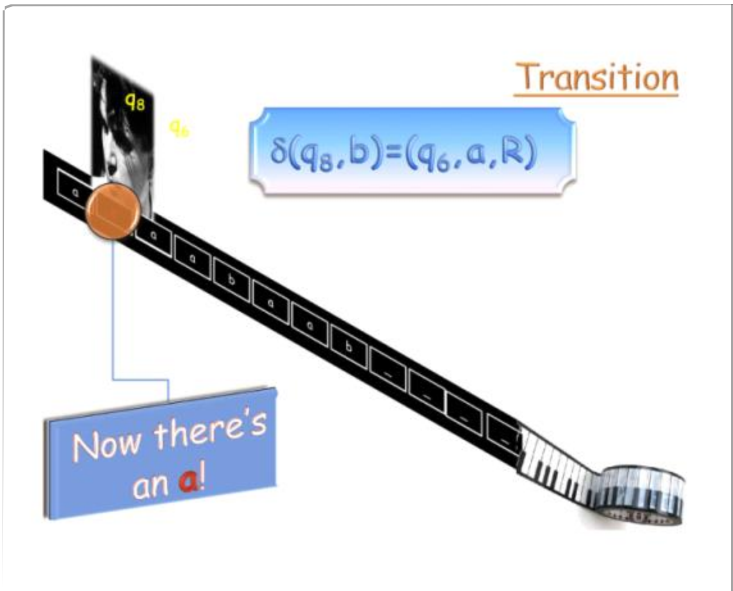
Here is how a Turing machine works:

- It has an infinite tape with letters from a given alphabet written on every cell.
- It has a reading head and a finite state machine.
- The read/write head, depending on the state the machine is in, can manipulate the cell it is looking at and then go either left or right.



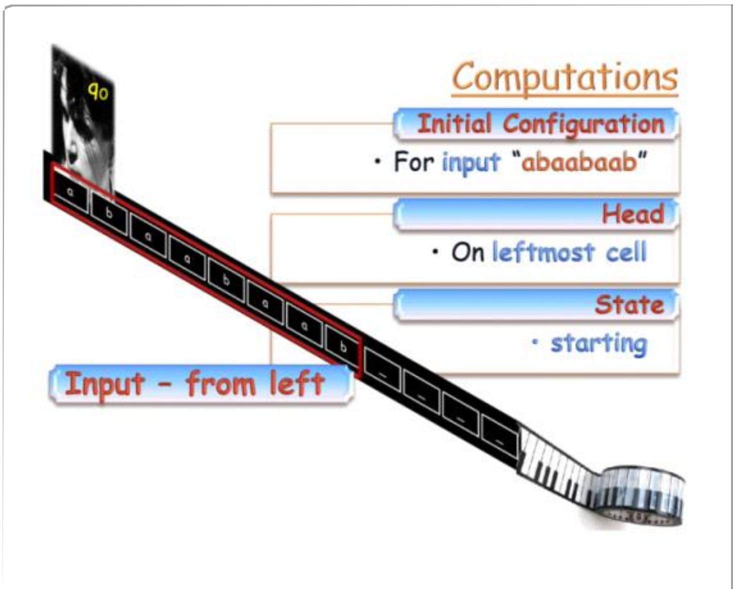
Formally, a Turing Machine's description consists of the following:

- A set of possible states the machine can be in
- The alphabet of the input
- Its extension the tape alphabet,
- The transition function, which determines which action to take in the next step,
- The state the machine starts in,
- The accepting state,
- And the rejecting state (there could be more than one such state).



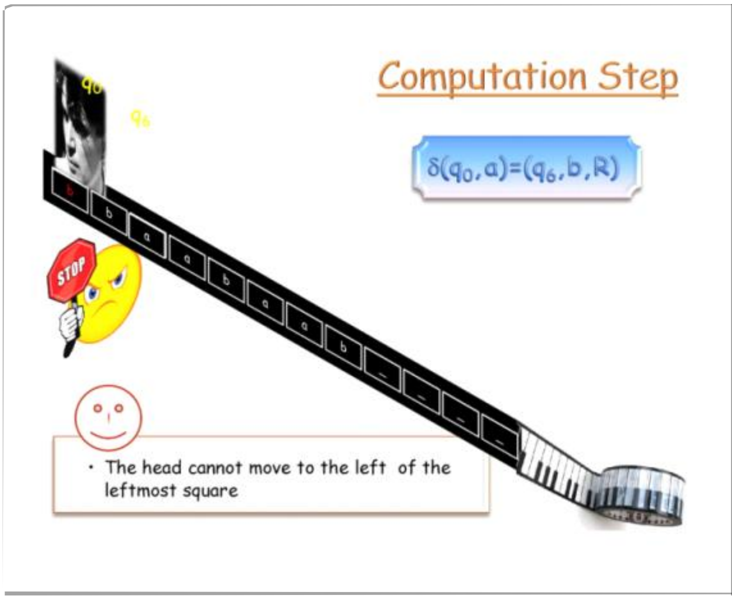
The transition function determines, for a given state and the content of the cell the read/write head is looking at:

- which <u>state</u> to be at <u>next</u>,
- which <u>letter</u> to <u>write</u> on that cell,
- and whether to go <u>right</u> or <u>left</u>.



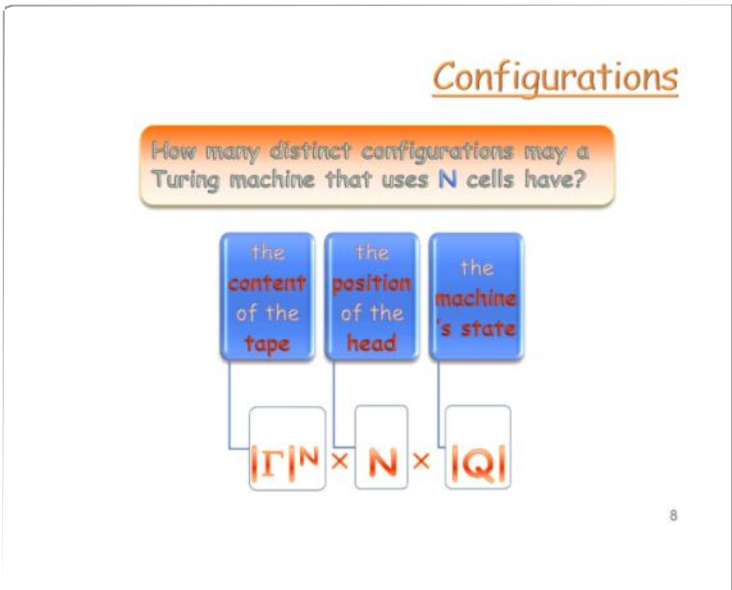
The computation starts with:

- The input written on the left most part of tape
- The head is on the left most cell.
- The state is starting state.



In computation steps, the transition function is applied to arrive at the new configuration.

We assume that the tape is infinite but only to the right direction.



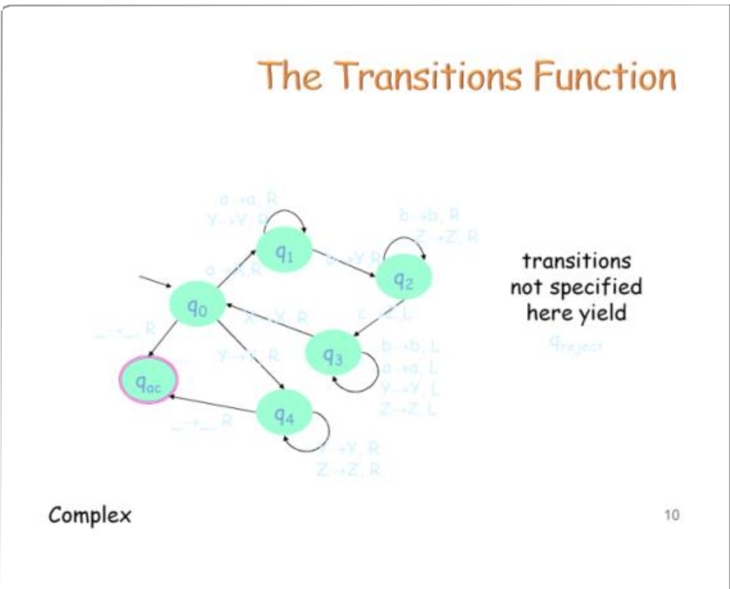
Let us now count the <u>number</u> of possible <u>configurations</u> a machine can have.

Assuming the machine uses only N cells of its infinite tape, we'd only consider:

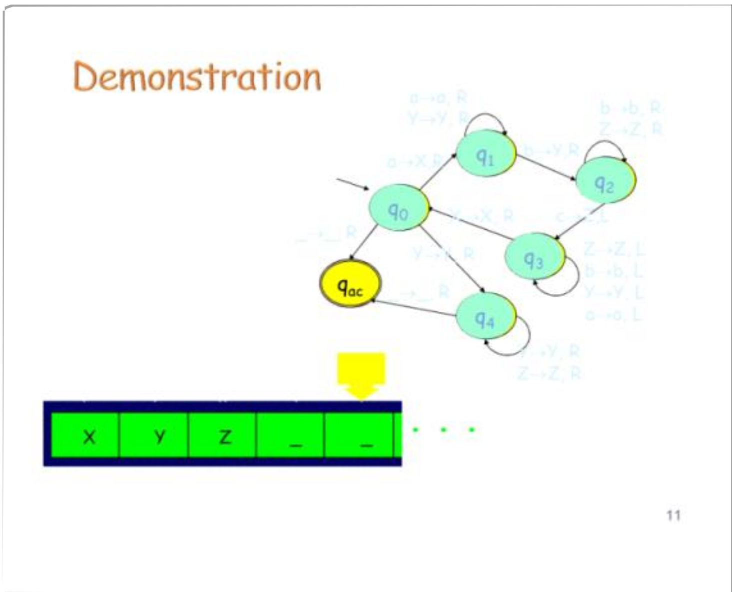
- The <u>content</u> of the tape,
- The <u>position</u> of the read/write head,
- And the machine's <u>state</u>.

Let us now consider a simple Turing machine for a language you're familiar with.

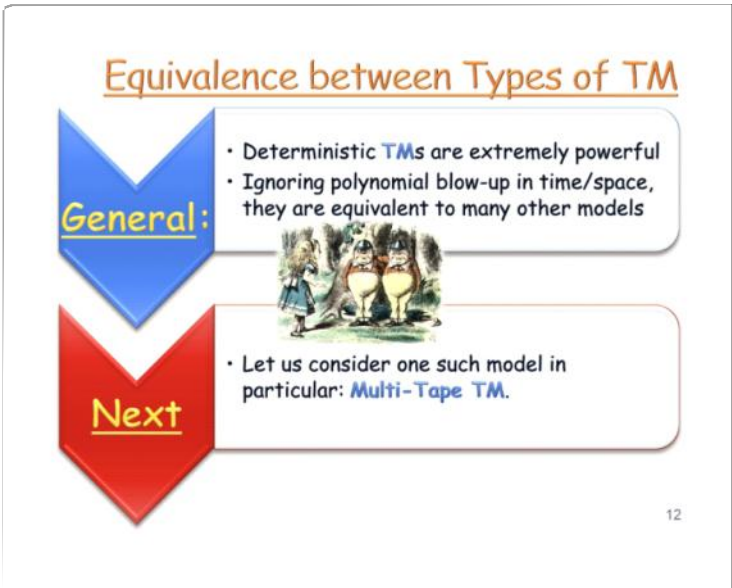
You have already seen that this language cannot be accepted by a finite automata, or by context free grammar.



Here is a description of the transition function for this machine.

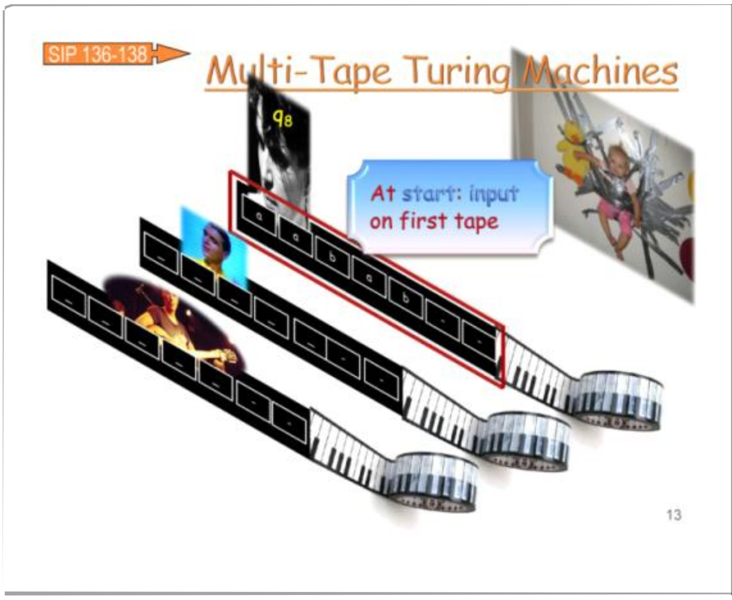


And here is a demo.



The deterministic TM model captures our notion of algorithm.

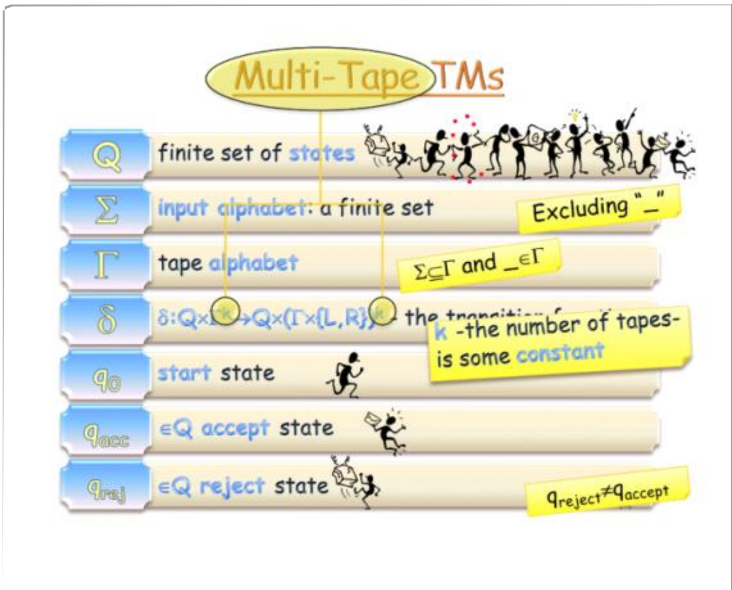
Next, we introduce a more general computational model of a multi tape deterministic TM,
and show it is equivalent to the deterministic TM model.



In a multi tape TM we have more than one tape, each with its own read/write head.

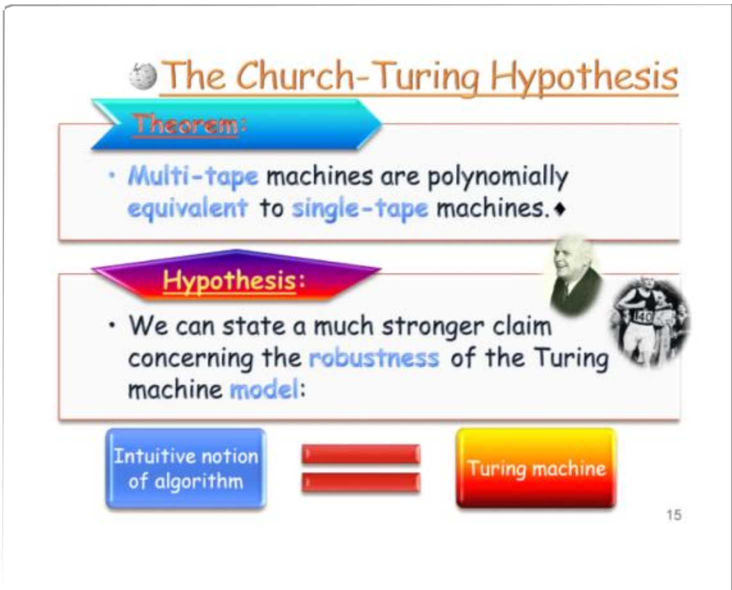
There is, however, only one state the machine is in at any given configuration!

In the starting configuration the input is written on the first tape.



Syntactically, the only difference between a regular TM and one with k tapes is in the transition function:

It takes as input k letters, and outputs k replacement letters, and k left/right instructions (plus a change in the machine's state) .

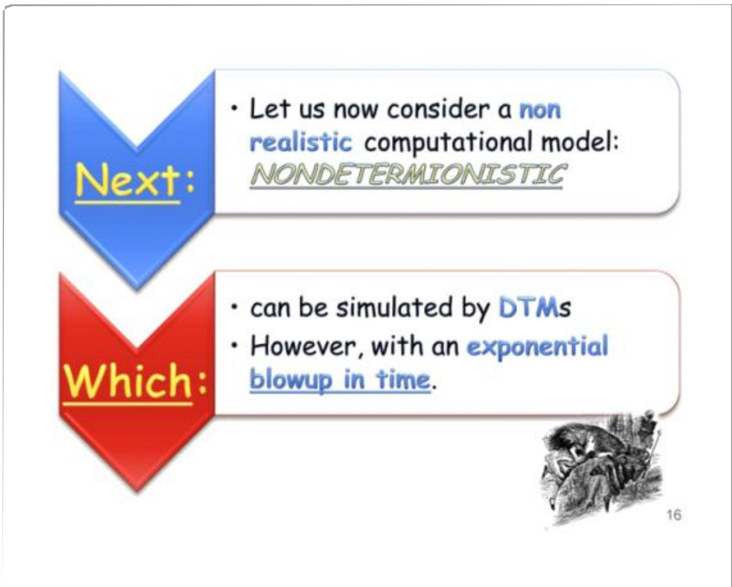


One can easily prove that these two models are equivalent.

The more general, obviously unproven hypothesis, suggested by Church and implicit in Turing's work, is that these models capture our intuitive notion of algorithm.

Some later models of computation may disagree with that hypothesis, in particular randomized or quantum algorithms;
we may discuss this late in the course.

Church-Turing Hypothesis

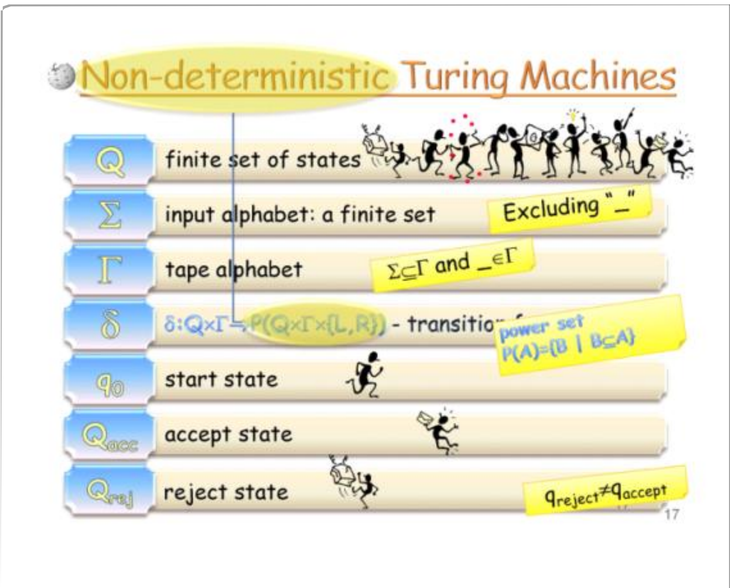


We now consider another variation of the TM model, however, one which does not at all correspond to any realistic notion of algorithm!

It is that of a non deterministic algorithm.

It can be translated into a deterministic one, however, with a huge blow up in time!

No better translation is known!



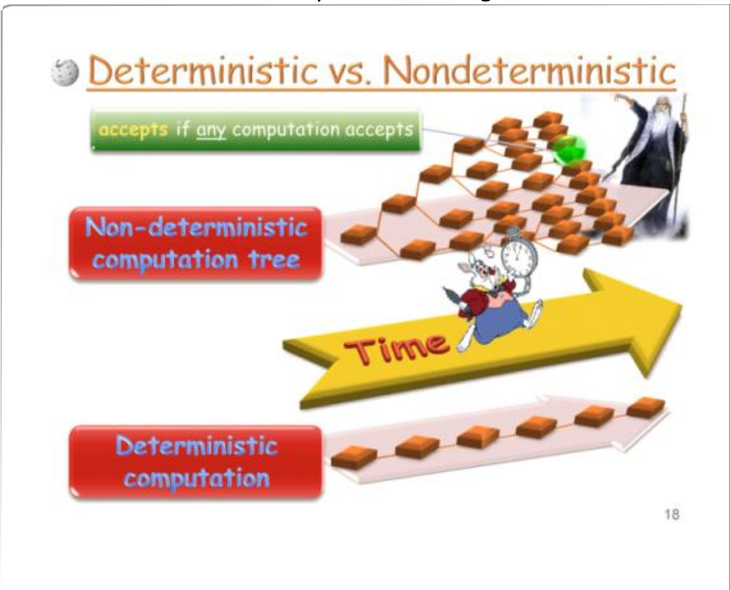
Syntactically, the difference between deterministic and non deterministic TM is only the transition function, which now becomes a transition relation.

For every state and letter it returns:

- A set of possible pairs of letter plus move, each of which is a possible computation step to take next.

[Non-deterministic Turing Machines]

A deterministic computation is a sequence of configurations each being the result of applying the transition function to the previous configuration.



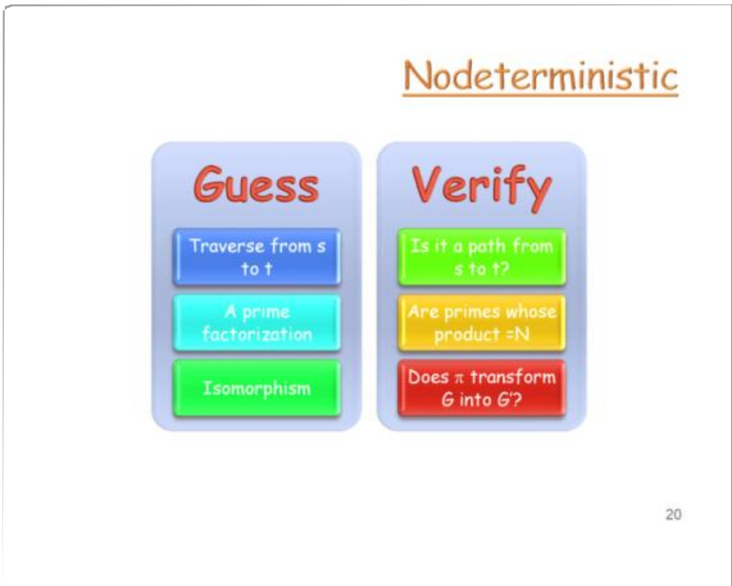
In a non deterministic computation there may be more than one transition possible from each configuration, which we can describe by a computation tree.

Time corresponds to the depths of the tree, hence the size of the tree may be exponential in the non deterministic running time.

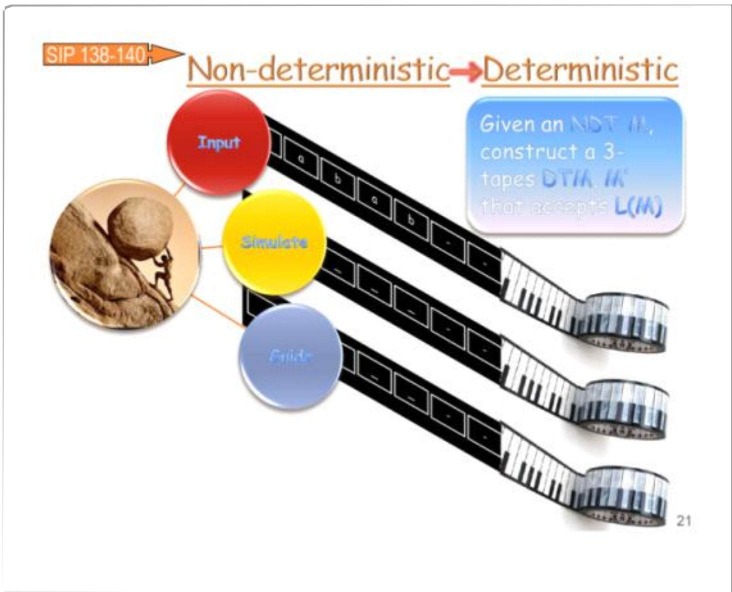
It suffices that one of the non deterministic computations (one of the paths in the non deterministic computation tree) accepts for the input to be accepted.

An alternative perspective of non deterministic computation is to think of it as a game between two players: one is magically powerful but untrustworthy; it tries to convince the other player, who has limited resources, that the input is in a given language L.

The first player sends the second player a witness that the input W is in the language L, which the second player has to verify efficiently.

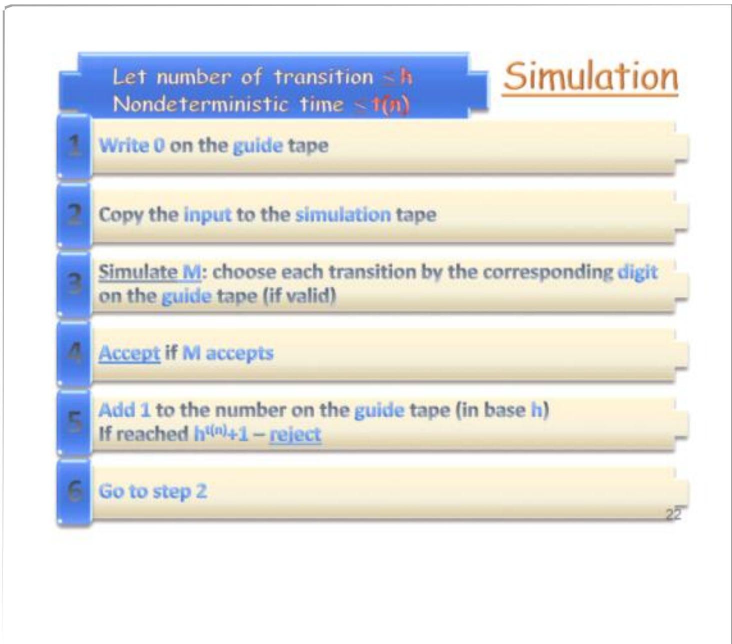


Here are a couple of examples of the two perspective on non deterministic algorithms.



Our simulation of non deterministic algorithms by deterministic ones uses three tapes:

- One for the input,
- one to guide us which computation path to take,
- And the third to simulate that computation.

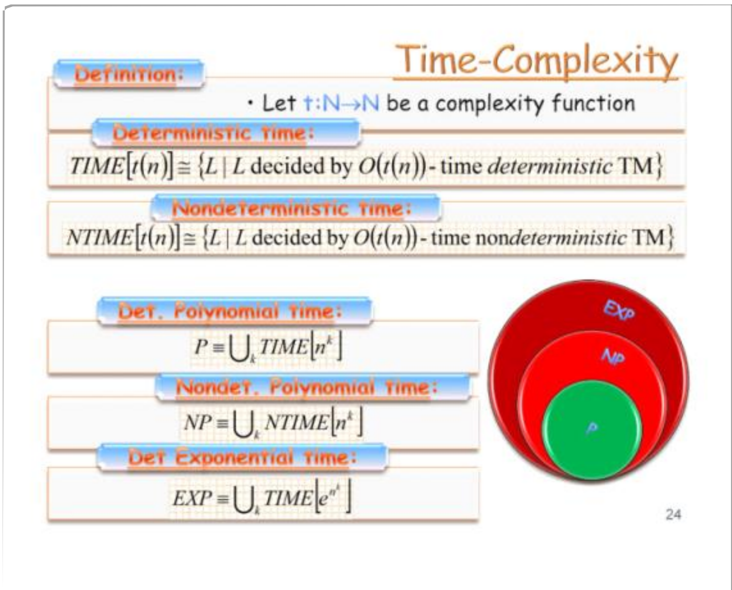


This simulation goes over all possible computations:

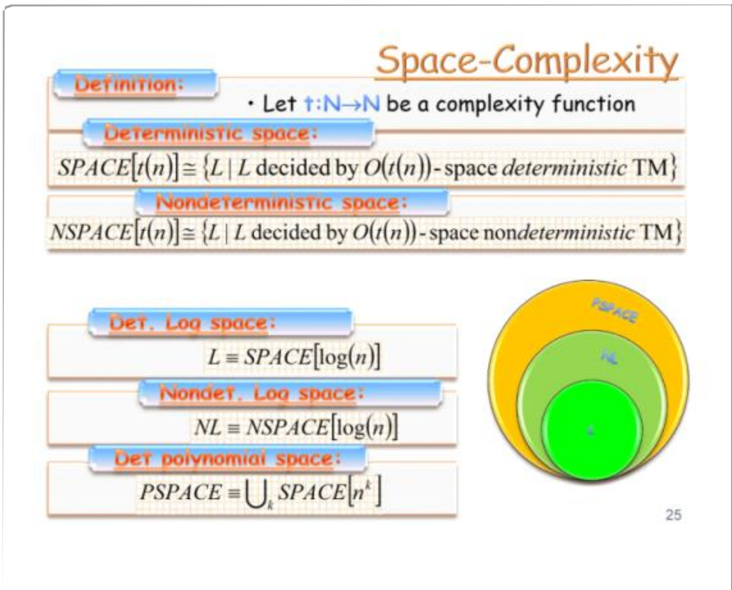
- If even one accepts, it accepts.
- If all rejects, it rejects.
- If one accepts, it accepts.
- If all rejects, it rejects.

Note that the time this simulation requires is exponential in the nondeterministic-time the machine runs in, t(n), and the space the simulation requires is O(t(n)).

With an agreed model of computation, by introducing bounds on resources required to compute the given problem, we are ready to define complexity classes.



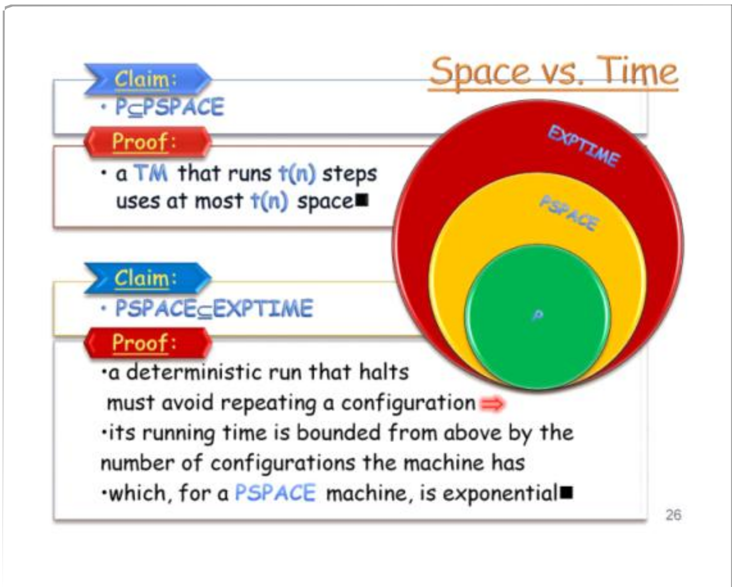
We can define time and non deterministic time complexity classes.



As well as space and non deterministic space complexity classes.

How can we define sub linear space classes where their input itself takes linear space?

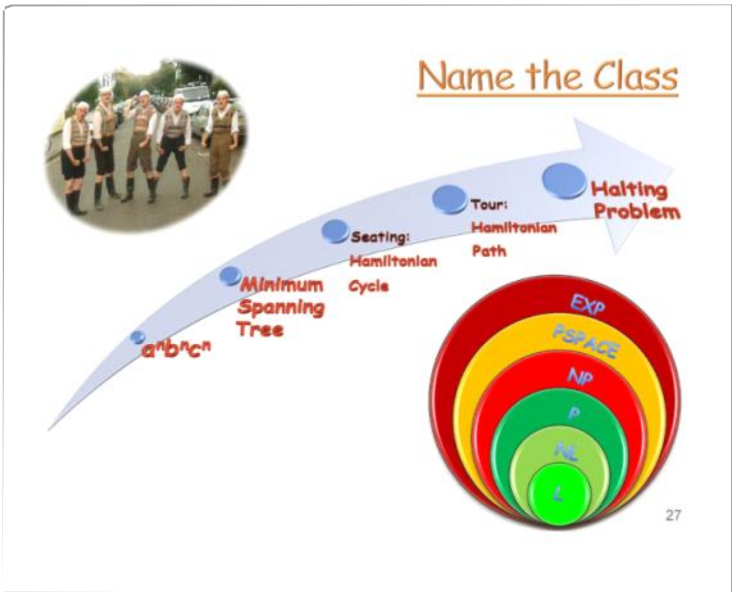
[Space Complexity]



Let us now prove very simple containments between time and space complexity.

Note that a deterministic machine that repeats a configuration twice must in fact be in an infinite loop.

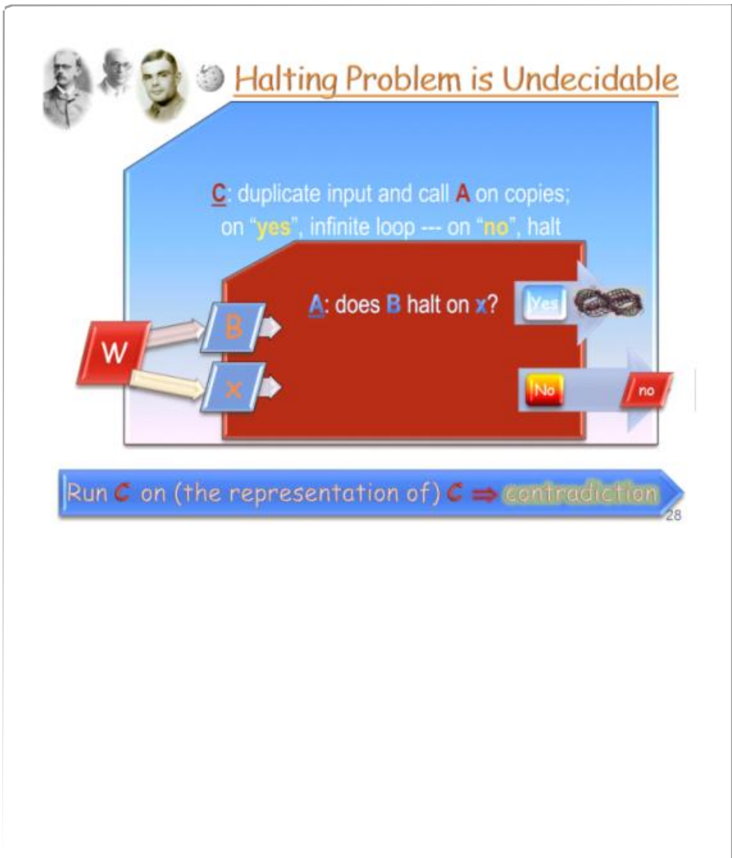
The number of configurations is essentially exponential in the number of cells the machine uses.



The picture presents a sequence of containments between complexity classes.
Are you sure they are all true?
Which ones can you prove at this point?
Are any of these containments strict?

Can you name the smallest class containing each of the problems?

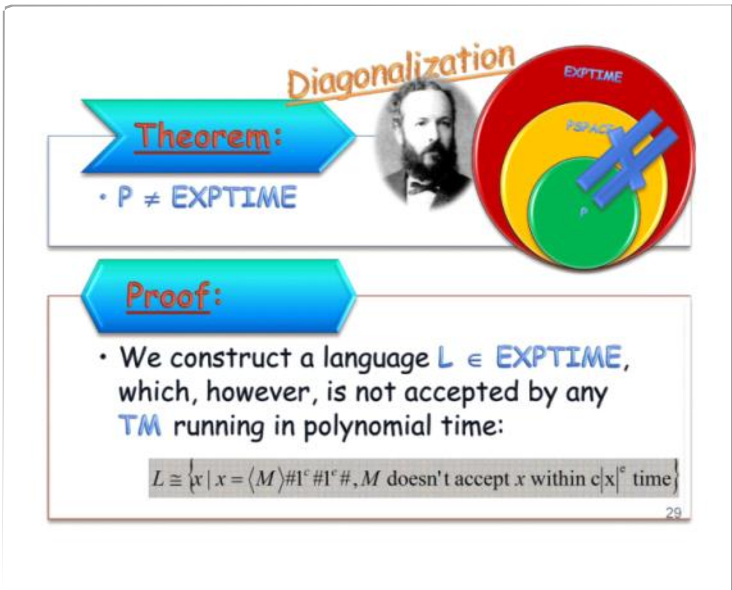
Alternatively, can you name the Gumby?



We would like to find out if some of these classes differ – so far it is quite possible all of the classes just defined are in fact the same.
Separating between classes of problems is not at all trivial -- there is, however, one prime example you most probably have studied before.

Let us now consider a simple proof that the halting problem is undecidable:
Assume by a way of contradiction a procedure A, that on inputs B and X, decides whether the procedure B halts on input X.
Construct procedure C that on input W, calls on procedure A with W being both B and X;
if A returns yes, C goes into an infinite loop, otherwise it stops and returns some answer.
Now run C with its input W being C itself: both options end up in a contradiction.

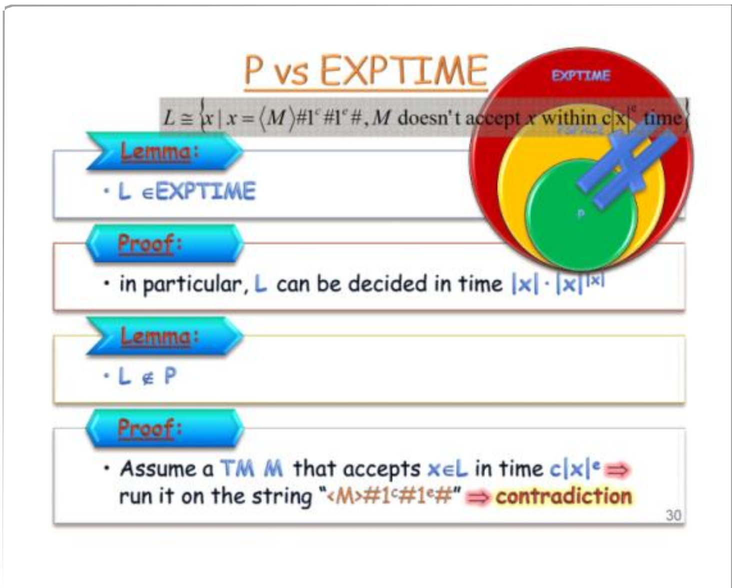
Halting Problem



We now apply a similar technique (known as Diagonalizing or self references and introduced by Cantor and Gödel) to show the class P is strictly smaller than the class EXPTIME.

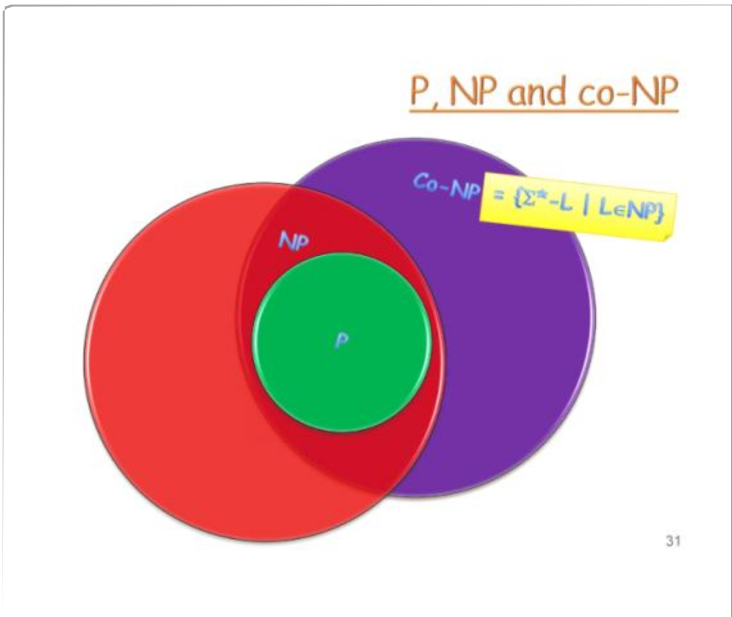
We construct a language in EXPTIME which, however, is not in P.
Inputs in that language must consist of a description of a TM and then two numbers written in unary.
The input is accepted if the machine described does not accept the input itself within the time specified by the two numbers.

[Diagonalization](#)



Their languages are in EXPTIME since a universal TM can simulate the computation of the machine on the input, and the running time is exponential (the two numbers written in unary).

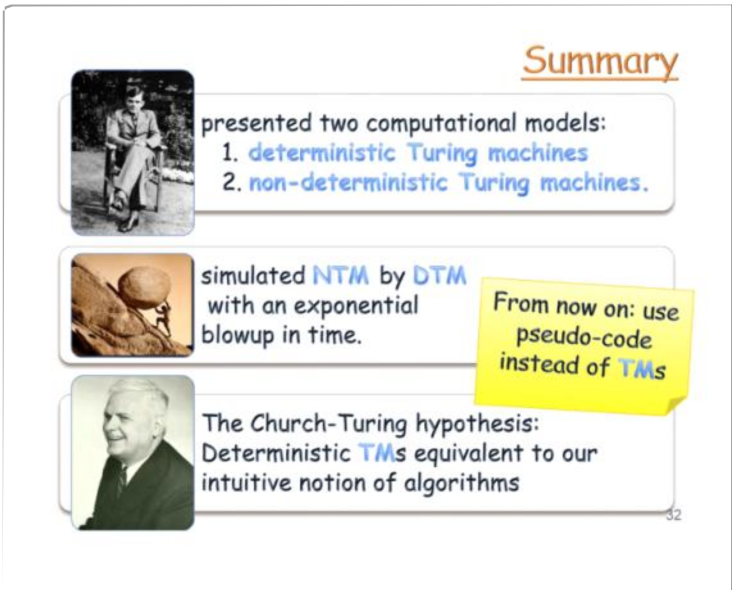
Assume by way of contradiction a TM with polynomial bounds on its running time that accepts L. Run it on its own description with the numbers corresponding to the bound of its running time.
Both possible outcomes result in a contraction.

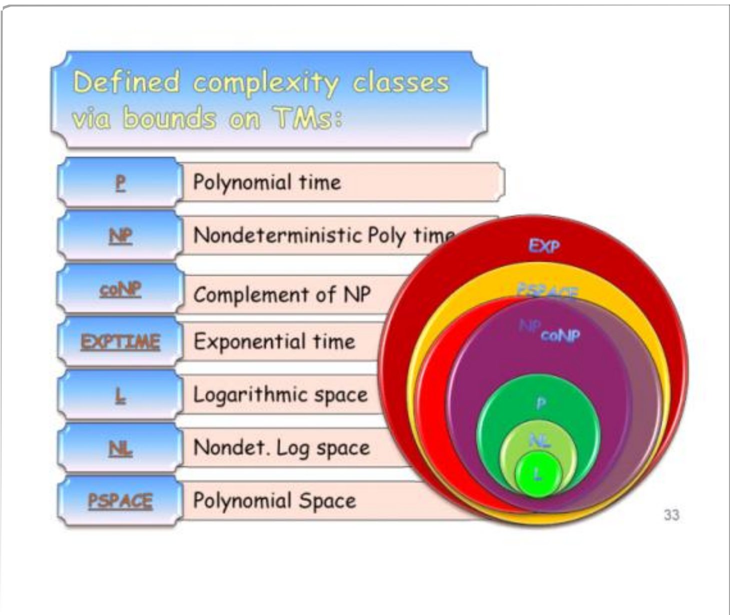


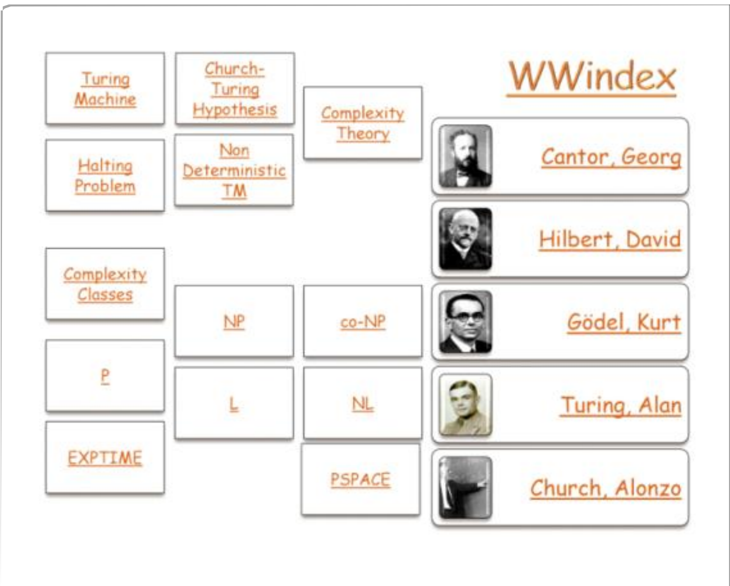
Let us now introduce a new class, coNP, which comprises all languages whose complement language is in the class NP.

The class P is clearly contained in both classes NP and coNP.

Can you prove any other relationships between these three classes?







- Turing Machine
- Church-Turing Hypothesis
- Complexity Theory
- Halting Problem
- Non Deterministic TM
- Complexity Classes
- P
- NP
- co-NP
- EXPTIME
- L
- NL
- PSPACE