

Machines

THE TIMES THEY ARE A - CHANGIN'

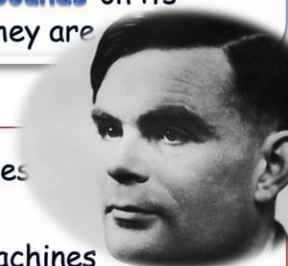
COME GATHER 'ROUND PEOPLE
WHEREVER YOU ROAM
AND ADMIT THAT THE WATERS

Goal:

- Complexity Theory classifies computational problems according to the amount of resources (say time) required
- Revisit the computational model "Turing Machine", this time discuss bounds on its resources and how robust they are

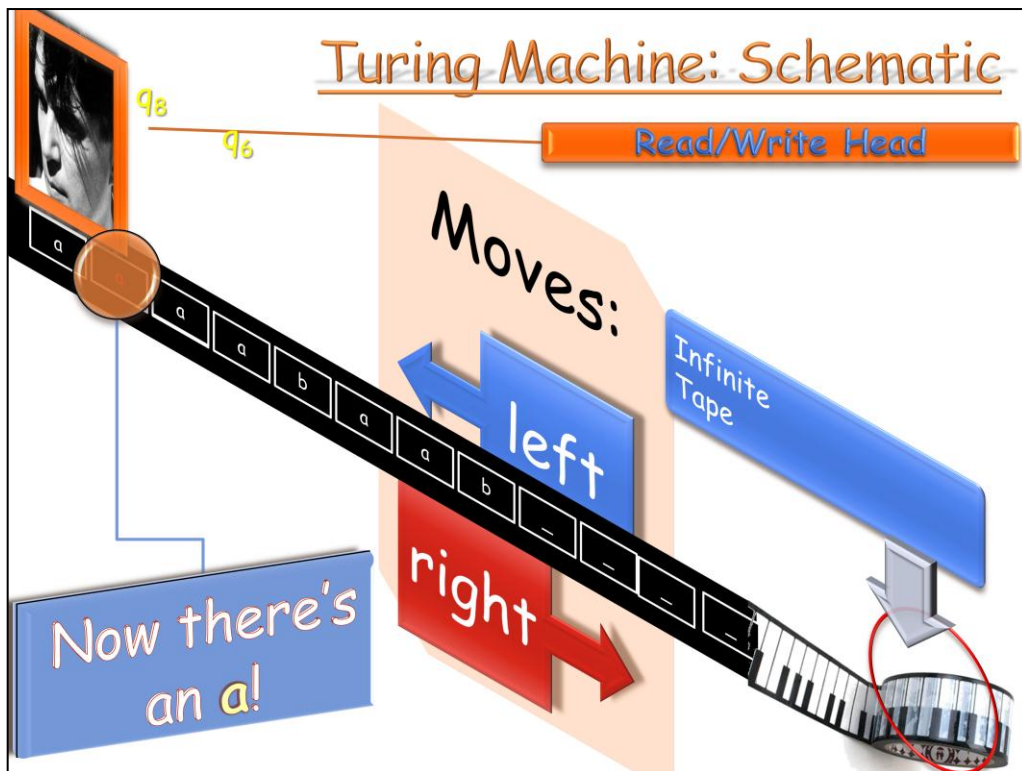
Plan:

- Deterministic Turing machines
- Multi-tape Turing machines
- Non-deterministic Turing machines
- The Church-Turing hypothesis
- Complexity classes as bounded TMs.



2

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 to introduce some complexity classes. We'll go over different types of Turing machines.



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.

TM: Formally

Syntactically, a TM consists of the following objects:

Q

finite set of states



Σ

input alphabet: a finite set

Excluding "_"

Γ

tape alphabet

$\Sigma \subseteq \Gamma$ and $_ \in \Gamma$

δ

$\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ - the transition function

q_0

start state



q_{acc}

$\in Q$ accept state



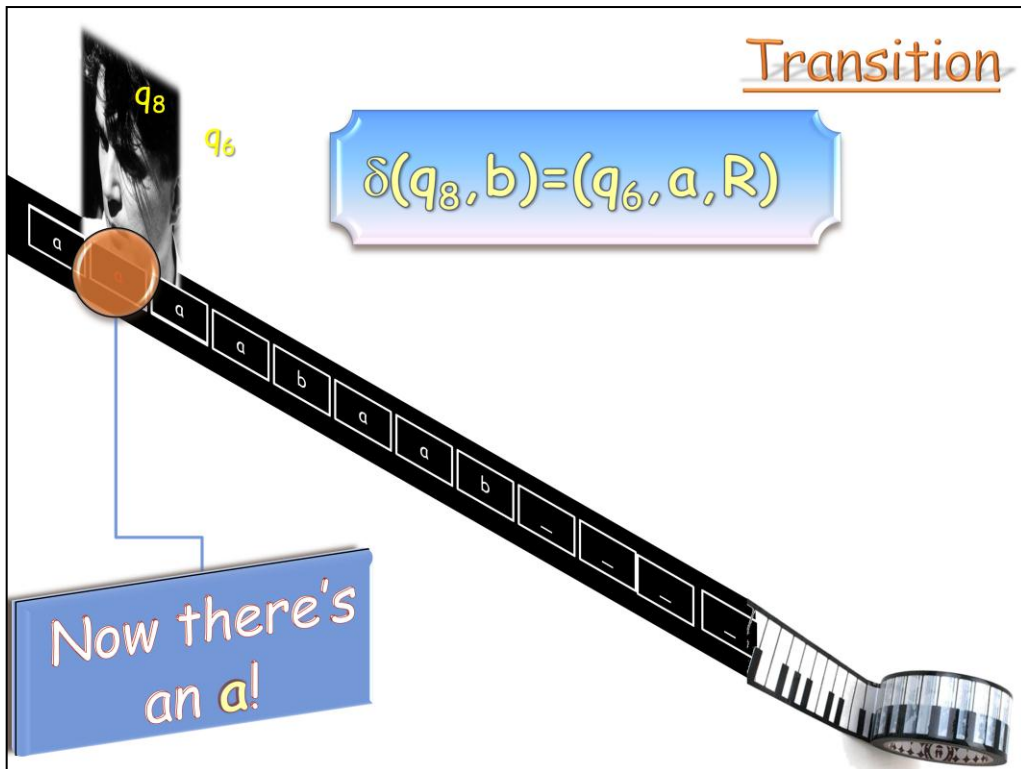
q_{rej}

$\in Q$ reject state

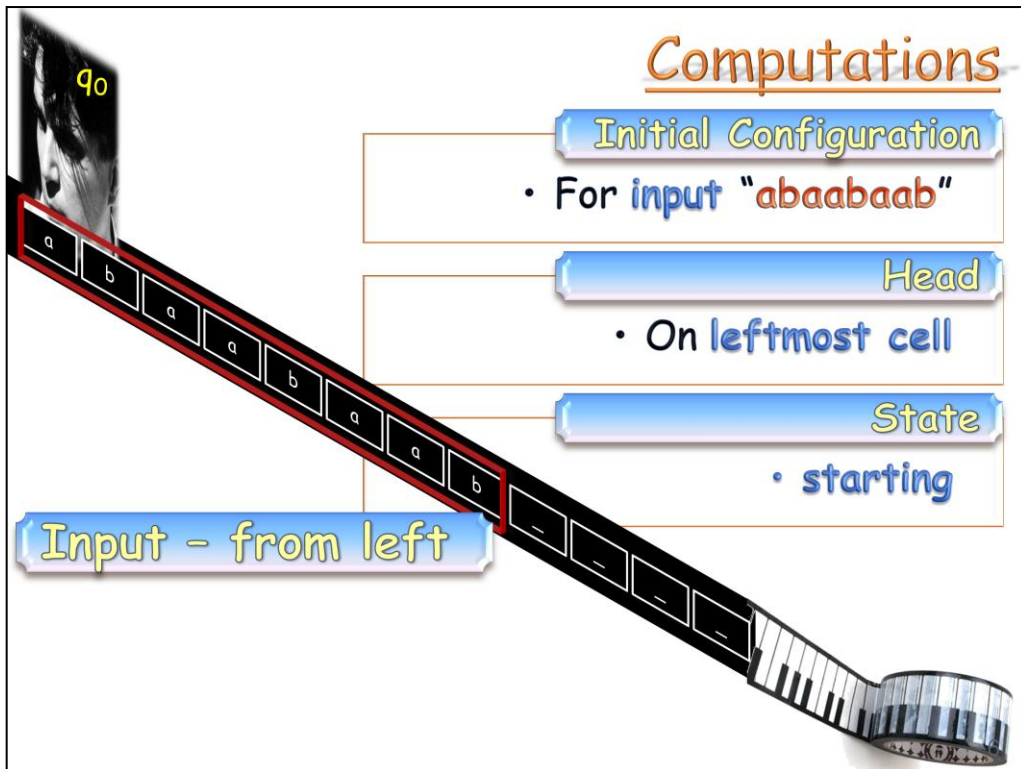


$q_{reject} \neq q_{accept}$

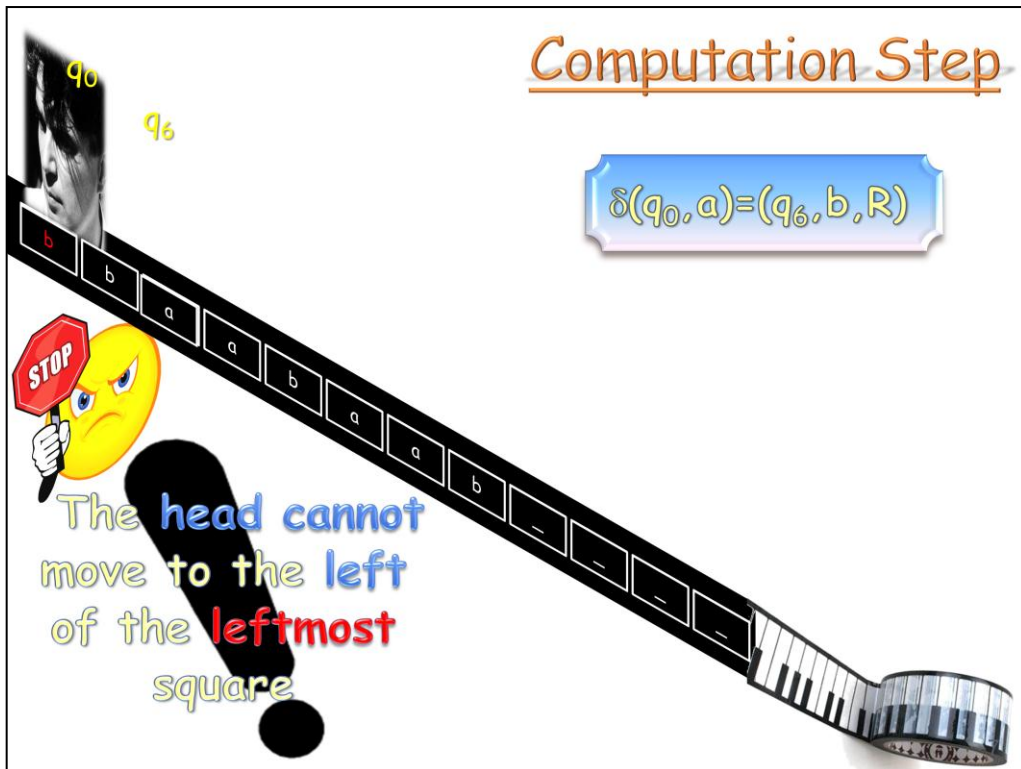
Formally, a Turing machine description consists of the following items: a set of possible states the machine can be in, the alphabet of the input, its extension to the alphabet of the tape, the transition function which determines which action to take in the next step, the state the machine starts in, the starting 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 state to be at next, which letter to write on that cell, and whether to go right or left.



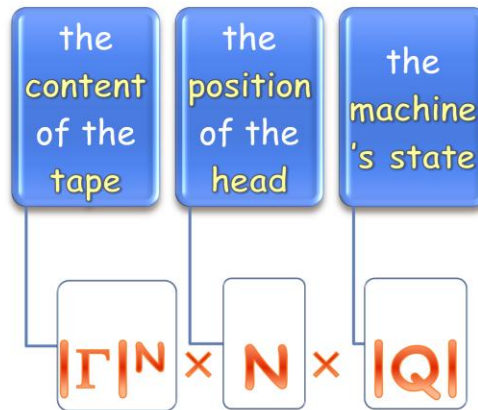
The computation starts with the input written on the left most part of tape.
The head is on the left most cell. The state is starting state.



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.

Configurations

How many distinct **configurations** may a Turing machine that uses **N** cells have?



8

Let us now count the number of possible configurations a machine can have: assuming the machine uses only N cells of its infinite tape, we'd only consider the content of the tape, the position of the read/write head, and the machine's state.

$L = \{a^n b^n c^n \mid n \geq 0\}$

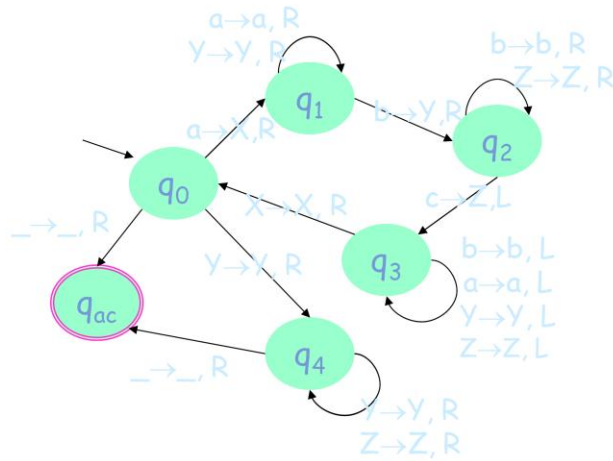
My first TM

Examples:
 Member of L: aaabbbccc
 Non-Member of L: aaabbbccc

Q	= $\{q_0, q_1, q_2, q_3, q_4, q_{\text{accept}}, q_{\text{reject}}\}$
Σ	= $\{a, b, c\}$
Γ	= $\{a, b, c, _, X, Y, Z\}$
δ	specified next...
q_0	- the start state. 
q_{acc}	$\in Q$ - the accept state. 
q_{rej}	$\in Q$ - the reject state. 

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.

The Transitions Function



transitions
not specified
here yield

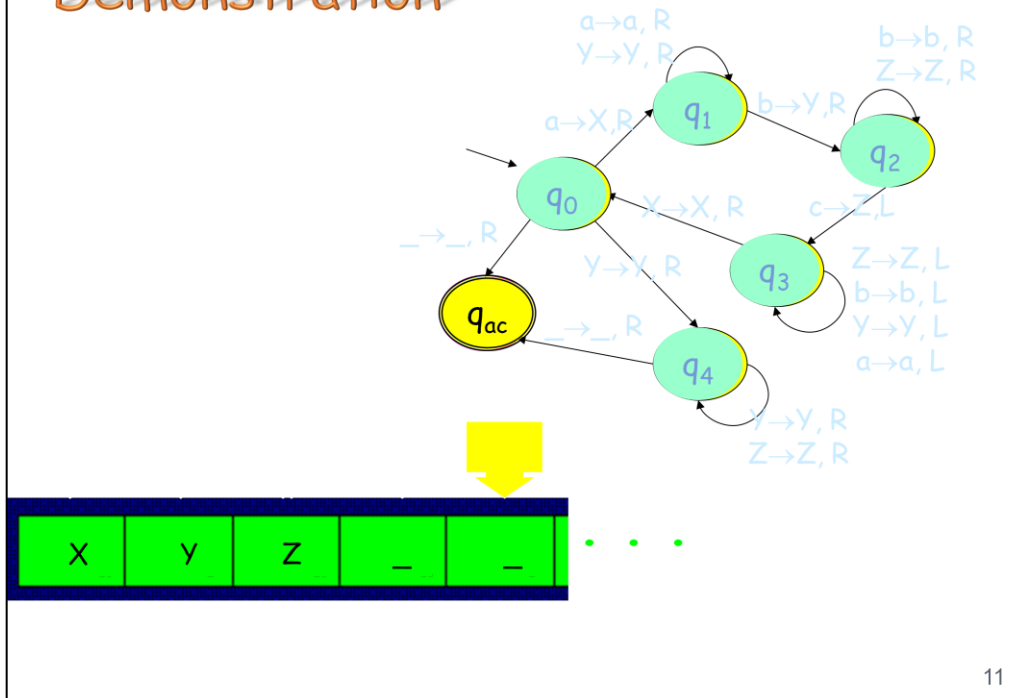
q_{reject}

Complex

10

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

Demonstration

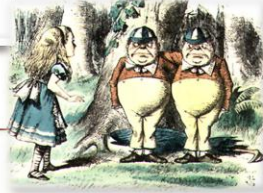


And here is a demo.

Equivalence between Types of TM

General:

- Deterministic **TM**s are extremely powerful
- Ignoring polynomial blow-up in time/space, they are equivalent to many other models



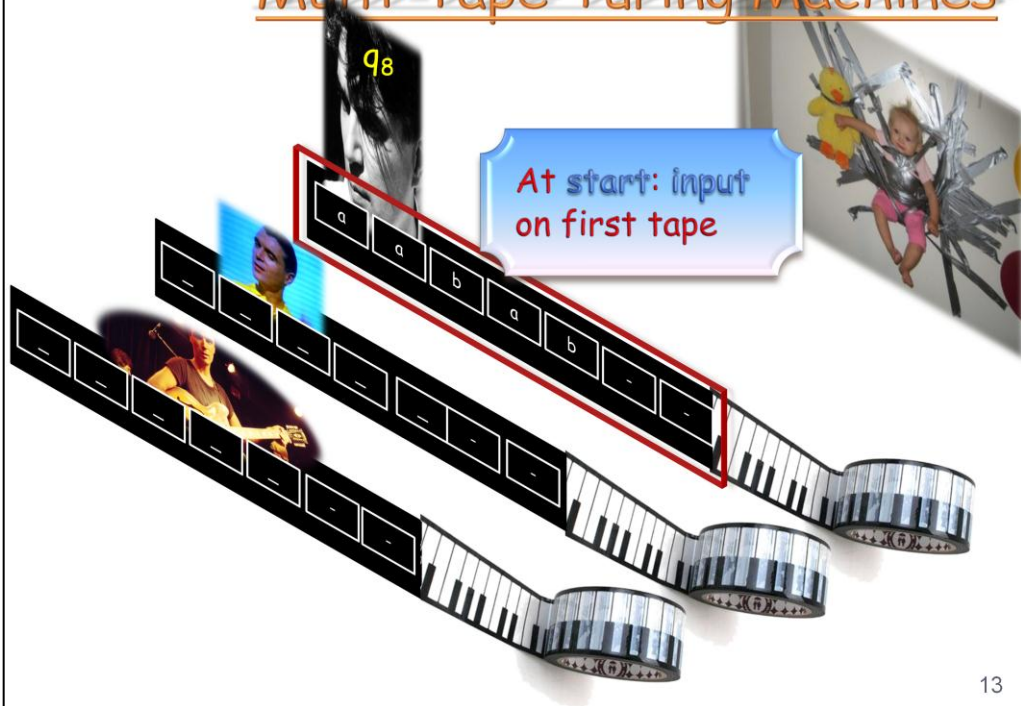
Next

- Let us consider one such model in particular: **Multi-Tape TM**.

12

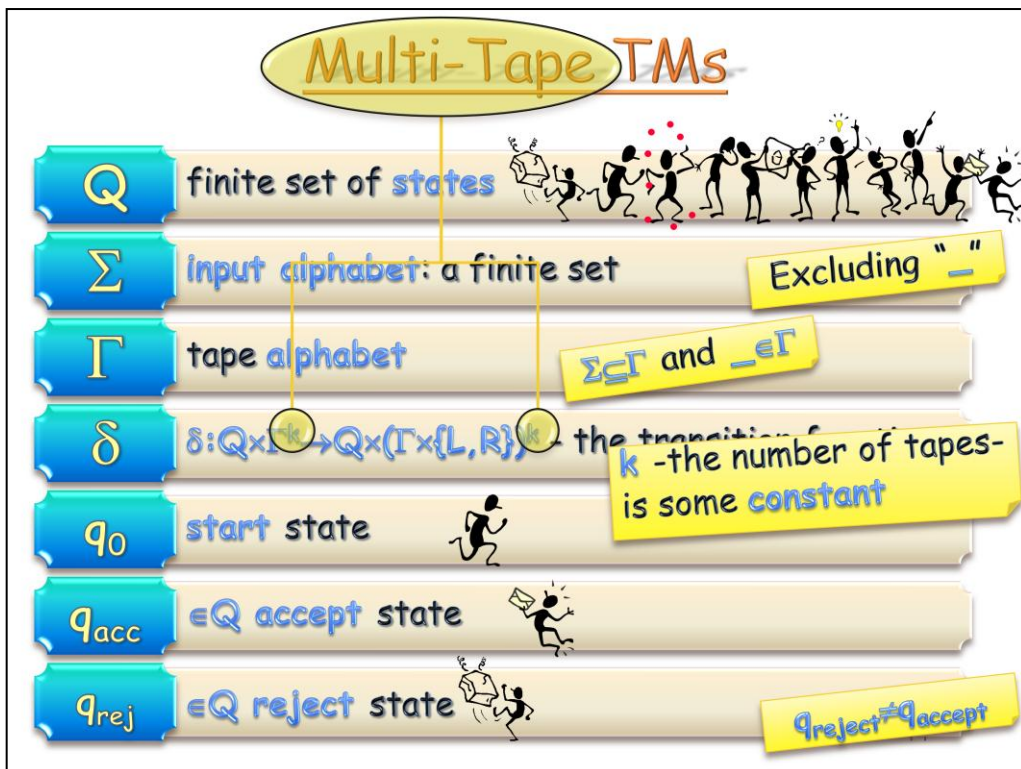
That deterministic TM model captures our notion of algorithm. Next, we introduce a more general computational model of a multi tape deterministic TM, and show it is equivalent to the deterministic TM model.

Multi-Tape Turing Machines



13

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 or right instructions (plus a change in the machine's state) .



The Church-Turing Hypothesis

Theorem:

- Multi-tape machines are polynomially equivalent to single-tape machines. ♦

Hypothesis:

- We can state a much stronger claim concerning the robustness of the Turing machine model:



Intuitive notion
of algorithm



Turing machine

15

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 later in the course.

Next:

- Let us now consider a **non realistic** computational model: *NONDETERMINISTIC*

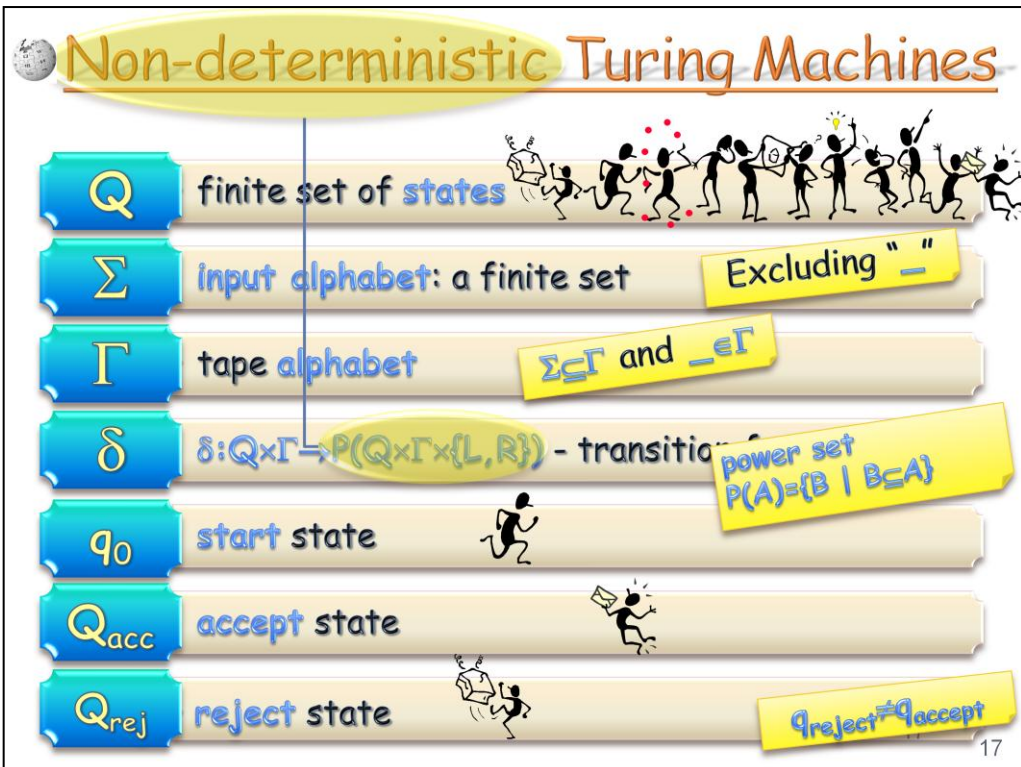
Which:

- can be simulated by **DTMs**
- However, with an **exponential blowup in time**.

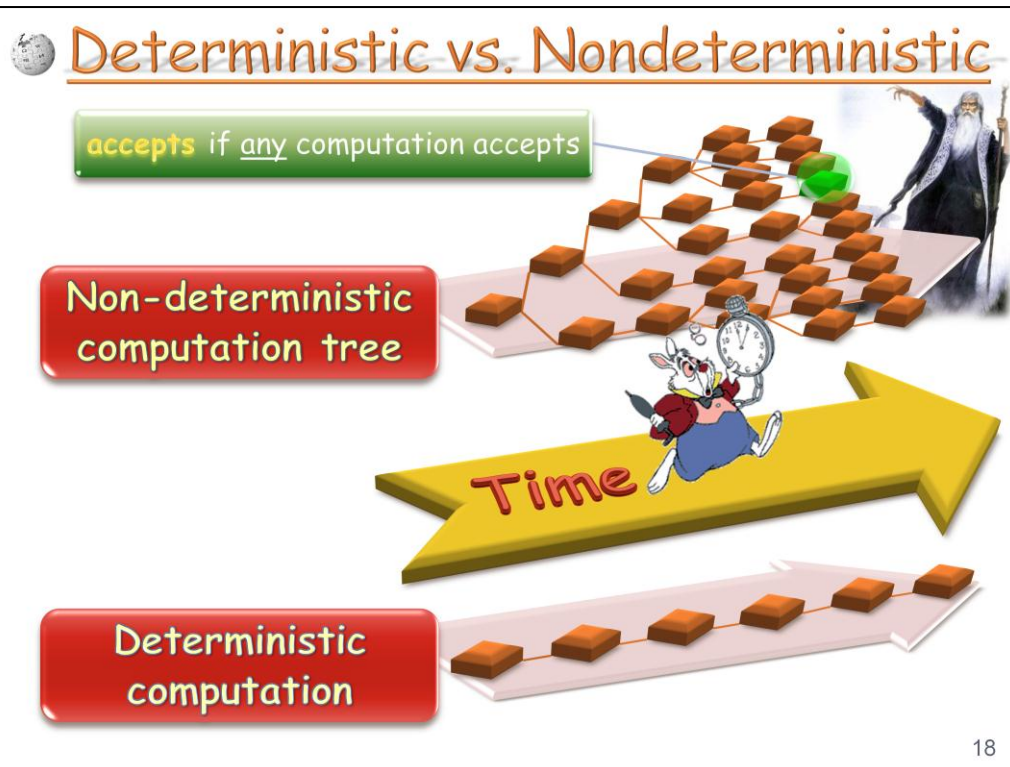


16

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.



18

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 that may be more than one transition possible from each configuration, which we can describe as 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.

Nondeterministic

Guess

Traverse from s
to t

A prime
factorization

Isomorphism

Verify

Is it a path from
 s to t ?

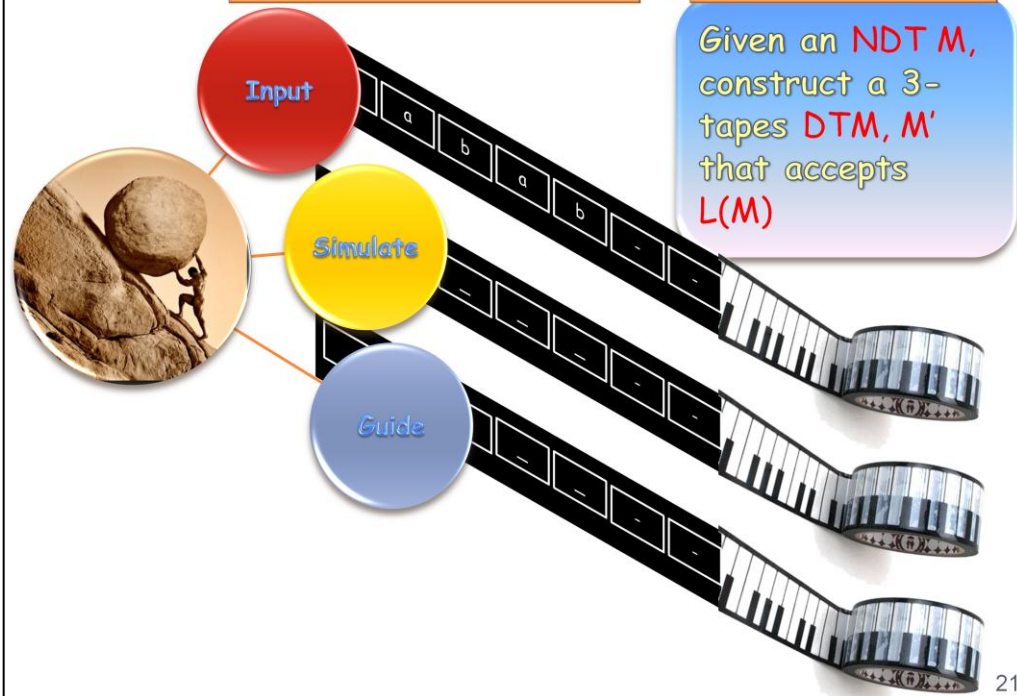
Are primes whose
product $=N$

Does π transform
 G into G' ?

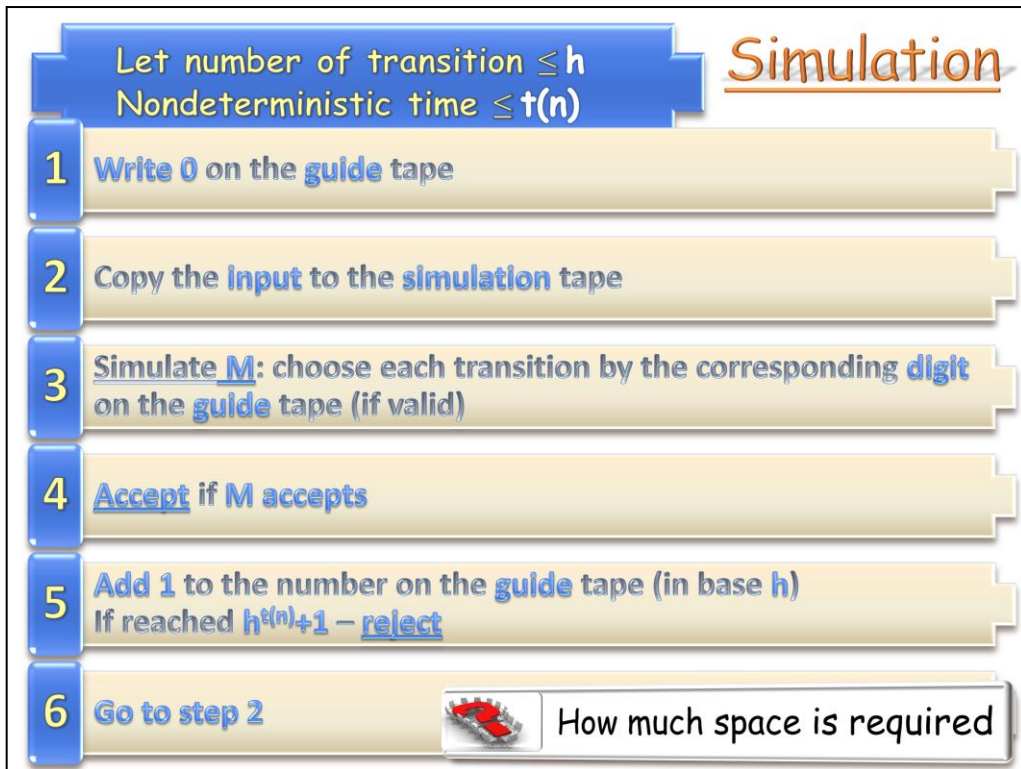
20

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

Non-deterministic → Deterministic

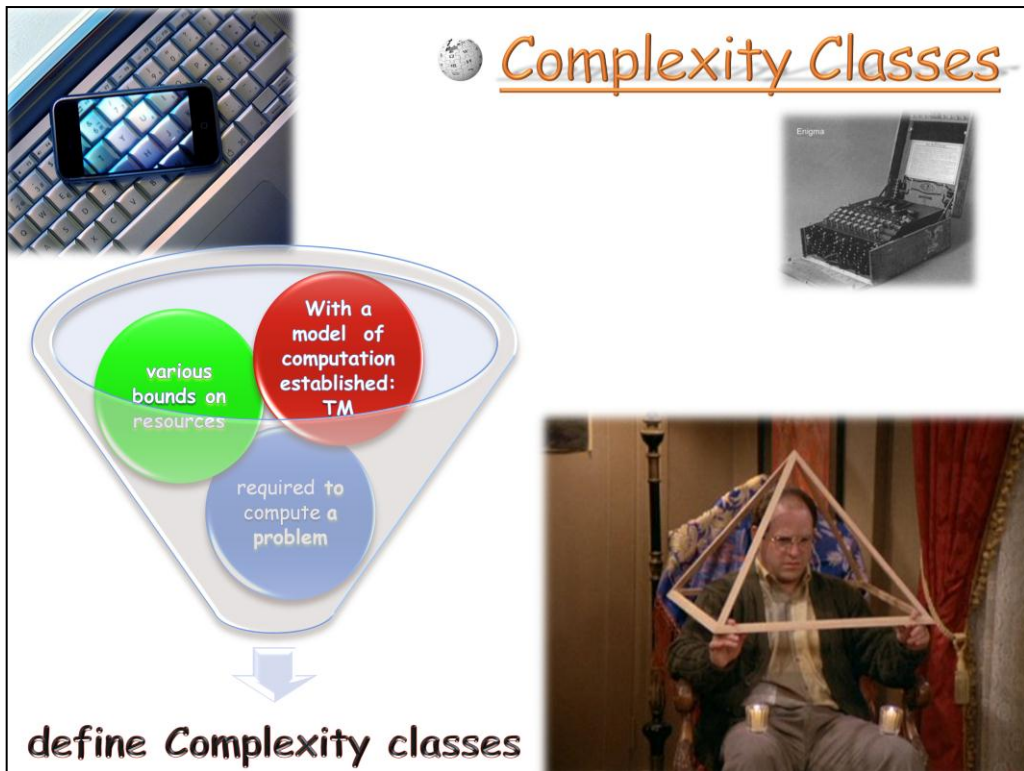


Our simulation of non deterministic algorithms by deterministic ones uses three tapes: one for the input, one to guide us which computation path to take, and the third to simulate that computation.



This simulation goes over all possible computations. If 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.

Time-Complexity

Definition:

- Let $t: \mathbb{N} \rightarrow \mathbb{N}$ be a complexity function

Deterministic time:

$TIME[t(n)] \cong \{L \mid L \text{ decided by } O(t(n))\text{-time deterministic TM}\}$

Nondeterministic time:

$NTIME[t(n)] \cong \{L \mid L \text{ decided by } O(t(n))\text{-time nondeterministic TM}\}$

Det. Polynomial time:

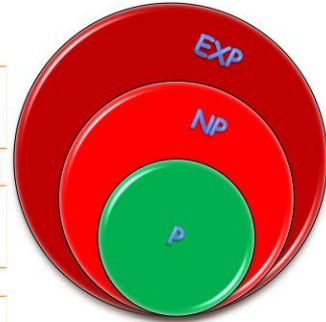
$$P \equiv \bigcup_k TIME[n^k]$$

Nondet. Polynomial time:

$$NP \equiv \bigcup_k NTIME[n^k]$$

Det Exponential time:

$$EXP \equiv \bigcup_k TIME[e^{n^k}]$$



24

We can define time and non deterministic time complexity classes.

Space-Complexity

Definition:

- Let $t: \mathbb{N} \rightarrow \mathbb{N}$ be a complexity function

Deterministic space:

$SPACE[t(n)] \equiv \{L \mid L \text{ decided by } O(t(n))\text{-space deterministic TM}\}$

Nondeterministic space:

$NSPACE[t(n)] \equiv \{L \mid L \text{ decided by } O(t(n))\text{-space nondeterministic TM}\}$

Det. Log space:

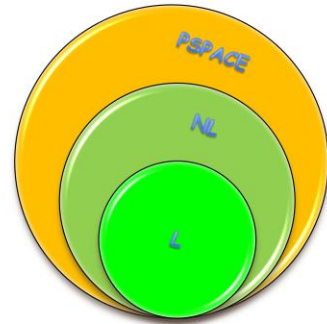
$$L \equiv SPACE[\log(n)]$$

Nondet. Log space:

$$NL \equiv NSPACE[\log(n)]$$

Det polynomial space:

$$PSPACE \equiv \bigcup_k SPACE[n^k]$$



25

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?

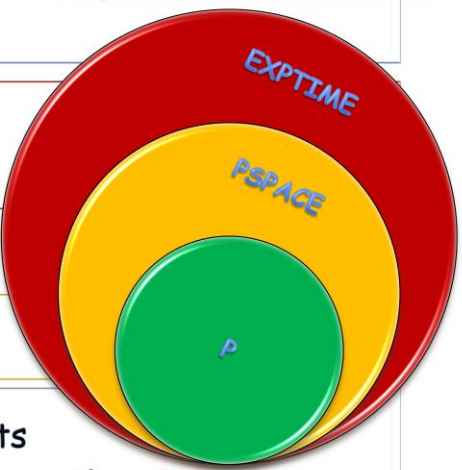
Space vs. Time

Claim:

- $P \subseteq PSPACE$

Proof:

- a TM that runs $t(n)$ steps uses at most $t(n)$ space ■



Claim:

- $PSPACE \subseteq EXPTIME$

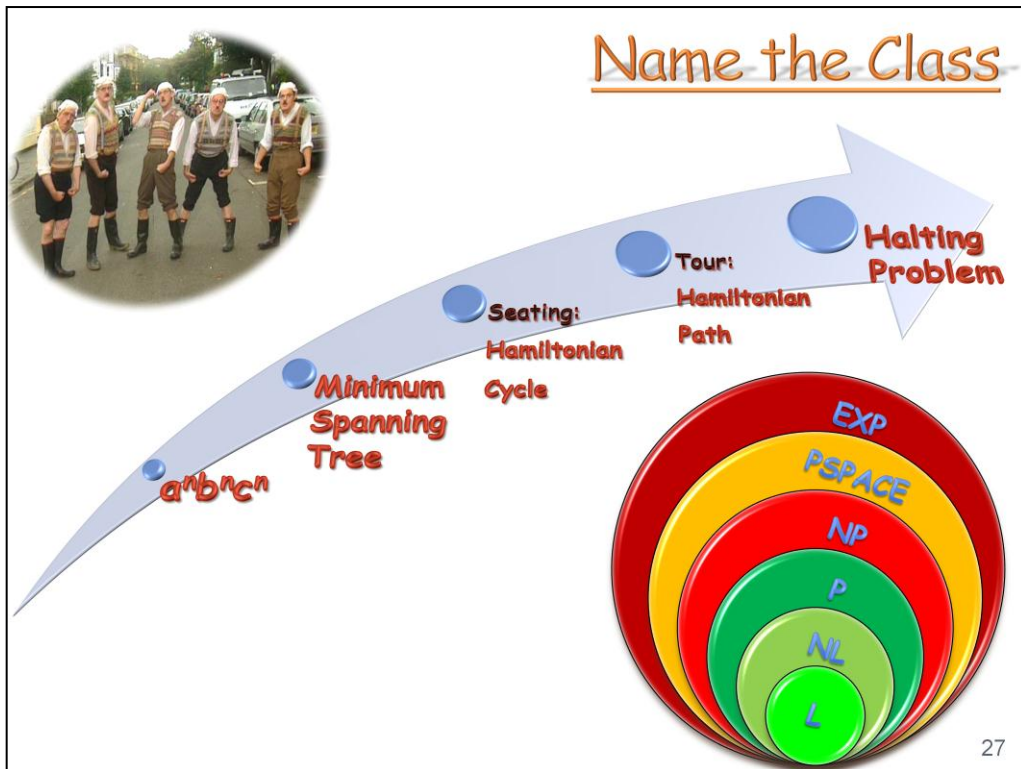
Proof:

- a deterministic run that halts must avoid repeating a configuration \Rightarrow
- its running time is bounded from above by the number of configurations the machine has
- which, for a PSPACE machine, is exponential ■

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.

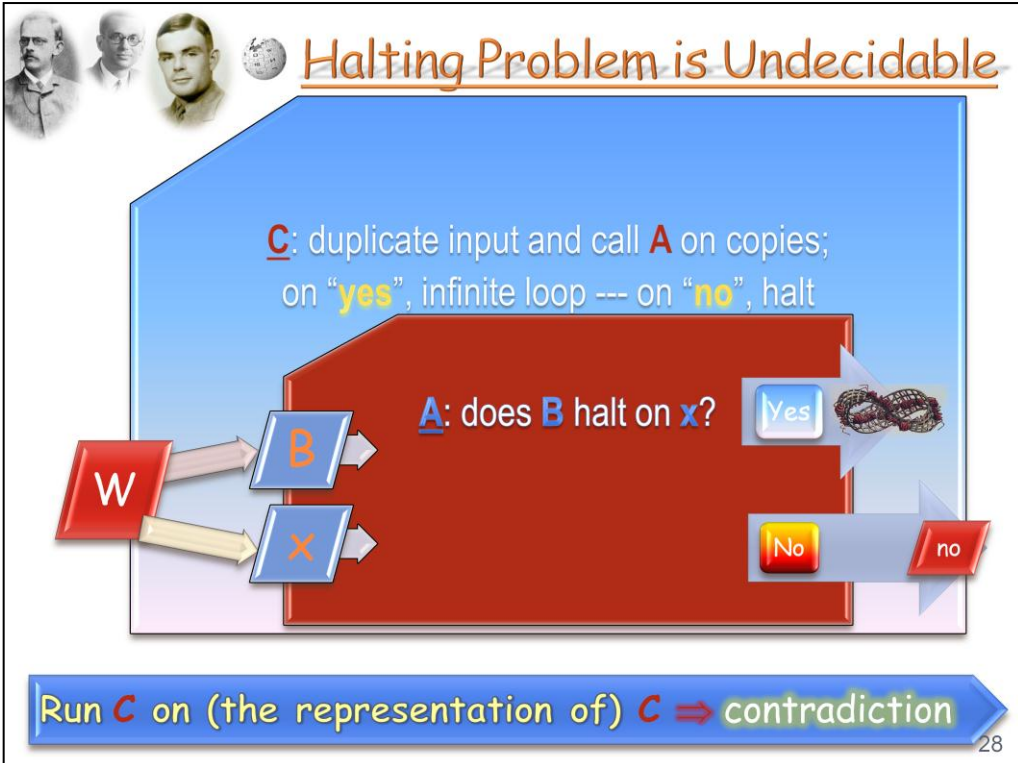
The number of configurations is essentially exponential in the number of cells the machine uses.



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:

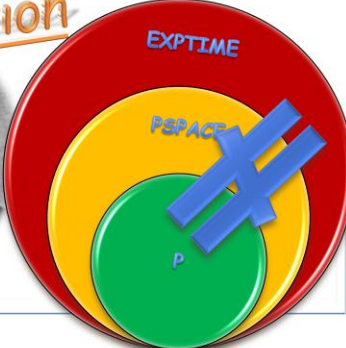
Assume by a way of contradiction a procedure A, that on inputs B and X, decides whether the procedure B halts on input X. Construct procedure C that on input W, calls on procedure A with W being both B and X; if A returns yes, C goes into an infinite loop, otherwise it stops and returns some answer.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

Diagonalization

Theorem:

- $P \neq EXPTIME$





Proof:

- We construct a language $L \in EXPTIME$, which, however, is not accepted by any TM running in polynomial time:

$$L \cong \{x \mid x = \langle M \rangle \# 1^c \# 1^e \#, M \text{ doesn't accept } x \text{ within } c|x|^e \text{ time}\}$$

29

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.

P vs EXPTIME

$L \cong \{x \mid x = \langle M \rangle \# 1^c \# 1^e \#, M \text{ doesn't accept } x \text{ within } c|x|^e \text{ time}\}$

Lemma:

- $L \in \text{EXPTIME}$

Proof:

- in particular, L can be decided in time $|x| \cdot |x|^{|x|}$

Lemma:

- $L \notin P$

Proof:

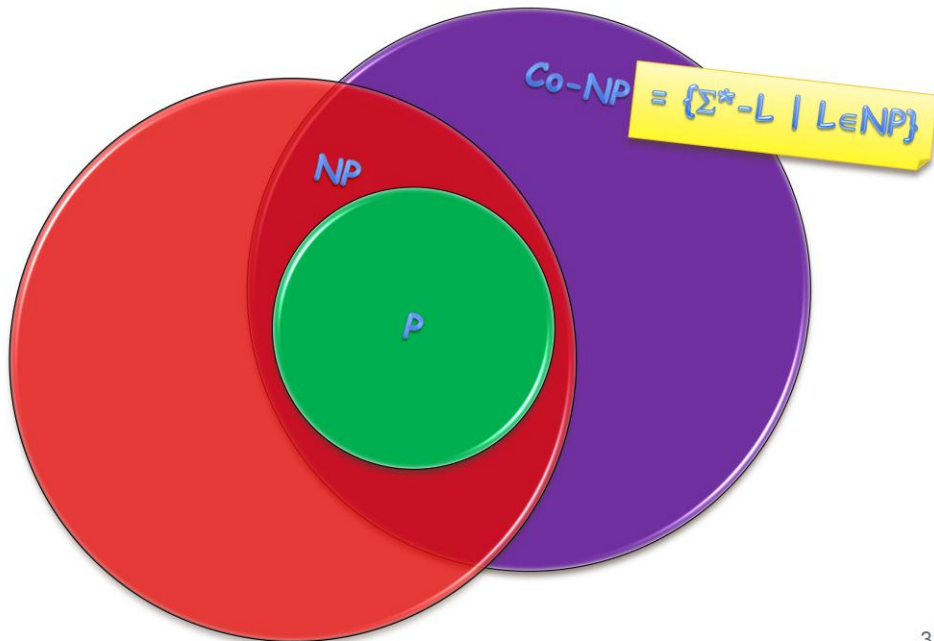
- Assume a TM M that accepts $x \in L$ in time $c|x|^e \Rightarrow$
run it on the string " $\langle M \rangle \# 1^c \# 1^e \#$ " \Rightarrow **contradiction**



30

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 contradiction.

P, NP and co-NP



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?

Summary



presented two computational models:

1. **deterministic Turing machines**
2. **non-deterministic Turing machines.**



simulated **NTM** by **DTM**
with an exponential
blowup in time.

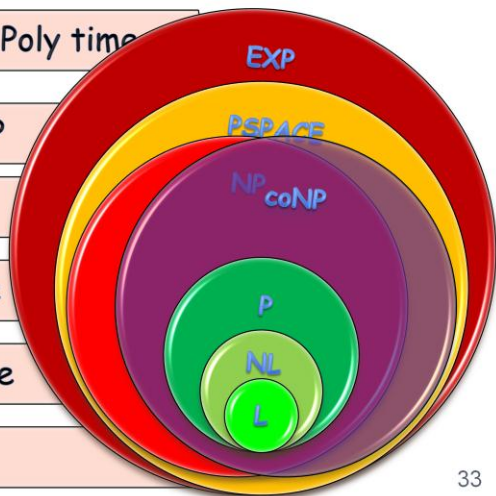
From now on: use
pseudo-code
instead of **TMs**



The Church-Turing hypothesis:
Deterministic **TMs** equivalent to our
intuitive notion of algorithms

Defined complexity classes via bounds on TMs:

<u>P</u>	Polynomial time
<u>NP</u>	Nondeterministic Poly time
<u>coNP</u>	Complement of NP
<u>EXPTIME</u>	Exponential time
<u>L</u>	Logarithmic space
<u>NL</u>	Nondet. Log space
<u>PSPACE</u>	Polynomial Space



33

