Computability and Complexity
COSC 4200

Conclusion

Main Questions of the Course

What problems can be solved on a computer?

What is a computer?

What is an algorithm?

Which problems can be solved on a computer? What is a computer?

Four types of algorithms – models of computation:

1. **Finite-State Automata**: algorithms with extremely limited memory
2. **Pushdown Automata**: algorithms that have a stack data structure
3. **Turing Machines**: any algorithm can be implemented on a Turing machine according to the *Church-Turing thesis*
4. **Polynomial-Time Algorithms**: efficient computation

Which problems can be solved on a computer? What is a computer?

Four types of algorithms – models of computation:

1. **Finite-State Automata**: algorithms with extremely limited memory

   equivalent in power to **regular expressions**

2. **Pushdown Automata**: algorithms that have a stack data structure

   equivalent in power to **context-free grammars**

3. **Turing Machines**: any algorithm can be implemented on a Turing machine according to the *Church-Turing thesis*

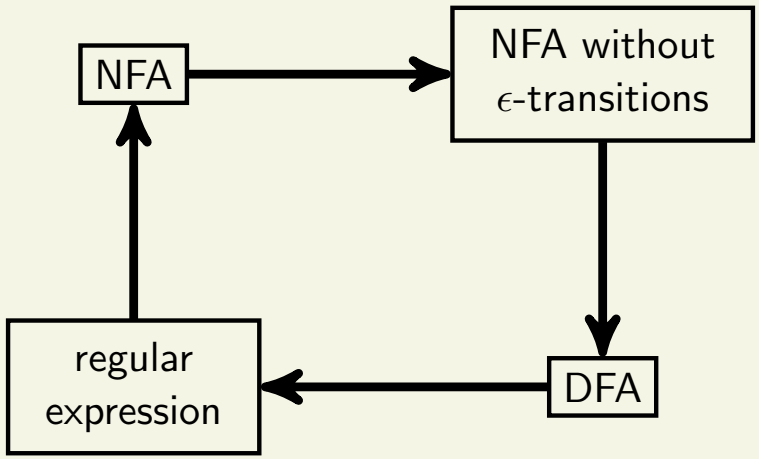
   We will study the **halting problem** *and* **undecidability***: many fundamental problems cannot be solved by any algorithm.*

4. **Polynomial-Time Algorithms**: efficient computation

   We will study **computational complexity** *and* **NP-completeness**. *Many optimization problems are (probably) impossible to solve efficiently on a computer.*

Finite Automata and Regular Languages

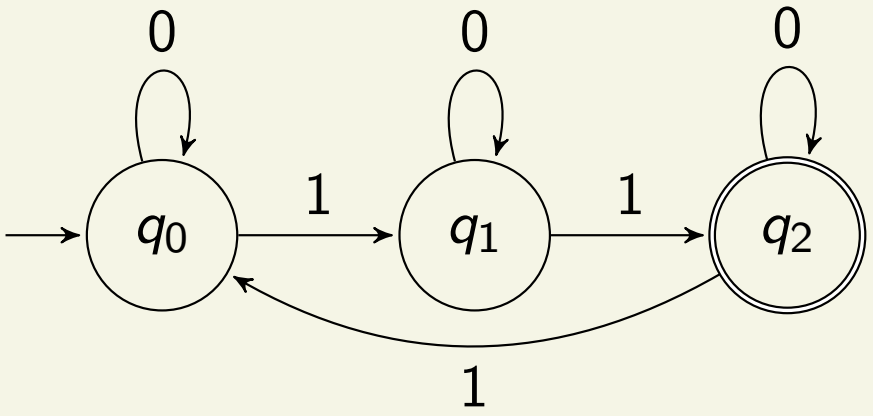
Regular Languages



Theorem

Let A be any language. The following are equivalent.

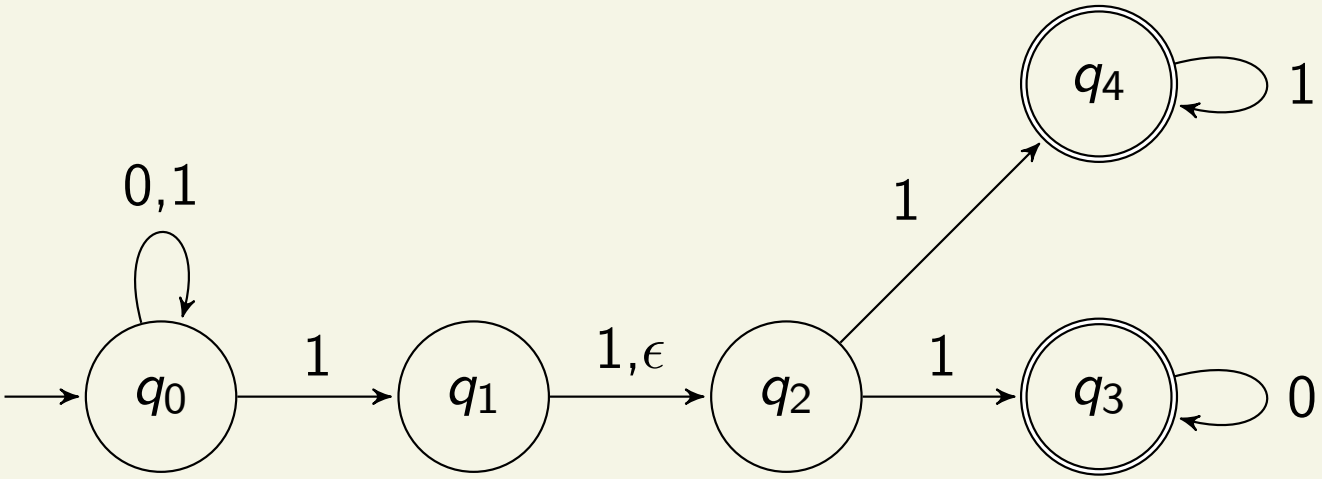
- *A is regular.*
- *$A = L(M)$ for some DFA M.*
- *$A = L(N)$ for some NFA N.*
- *$A = L(N')$ for some NFA N' without ϵ-transitions.*
- *$A = L(R)$ for some regular expression R.*



Definition

A *(deterministic) finite automaton* (often abbreviated *DFA*) is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$ where

- Q is a finite set of *states*
- Σ is an *alphabet*
- $\delta : Q \times \Sigma \to Q$ is the *transition function*
- $q_0 \in Q$ is the *initial state*
- $F \subseteq Q$ is the set of *accepting states*



Definition

A *nondeterministic finite automaton* (often abbreviated *NFA*) is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$ where

- Q is a finite set of *states*
- Σ is an *alphabet*
- $\delta : Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$ is the *transition function*
- $q_0 \in Q$ is the *initial state*
- $F \subseteq Q$ is the set of *accepting states*

Regular Expressions

A regular expression is built from alphabet symbols, ϵ, $\emptyset$, $\cup$, $\cdot$, $*$, and parentheses. Defined formally using induction:

Definition

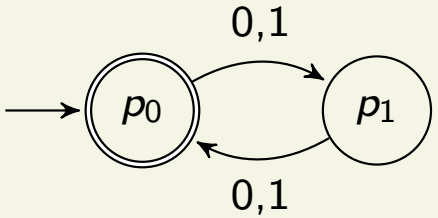
Let Σ be an alphabet. R is a *regular expression* if R is

1. a for some $a \in \Sigma$
2. ϵ
3. $\emptyset$
4. $(R_1 \cup R_2)$ where R_1 and R_2 are regular expressions
5. $(R_1 \cdot R_2)$ where R_1 and R_2 are regular expressions
6. (R_1^*) where R_1 is a regular expressions

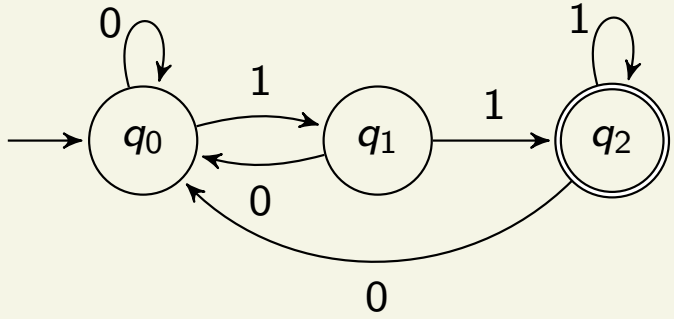
Often the parentheses and concatenation are omitted, if the meaning is clear.

Product Construction

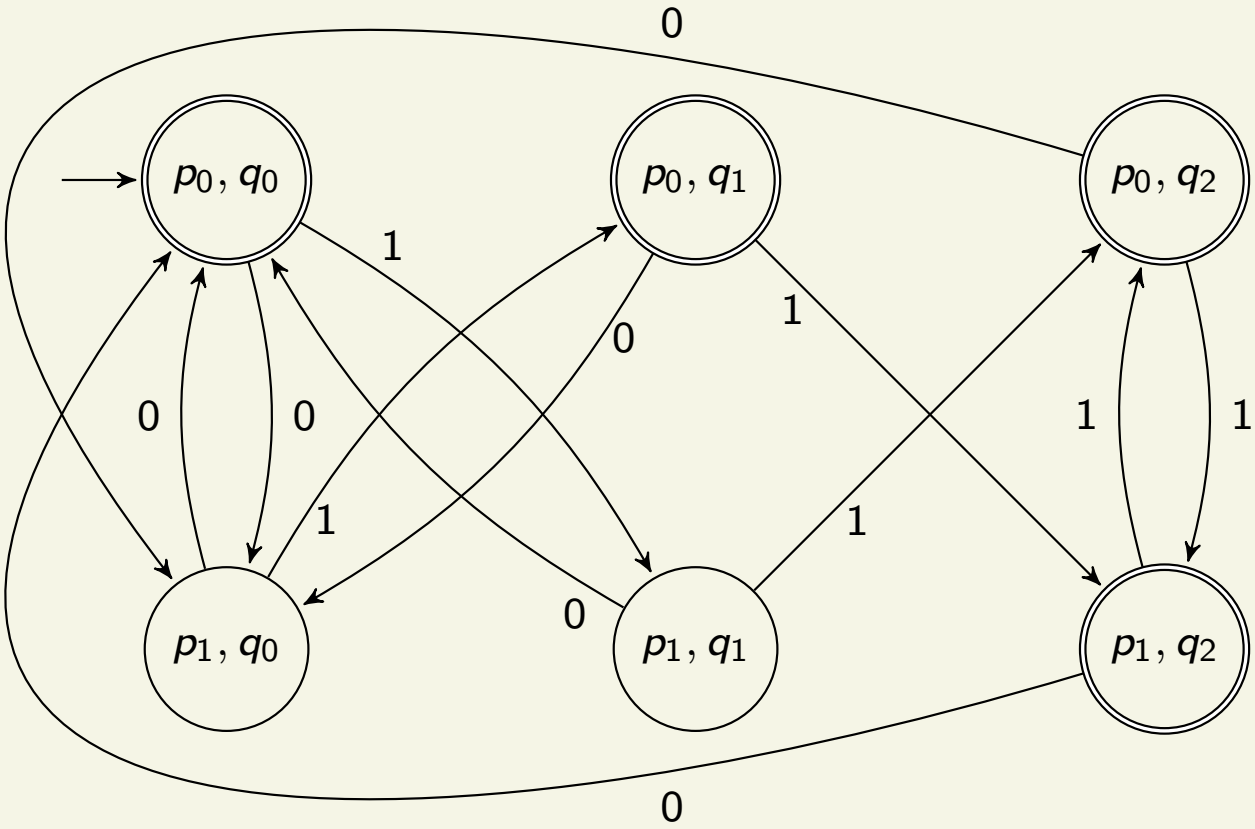
$A = \{x \in \{0,1\}^* \mid |x| \text{ is even}\}$



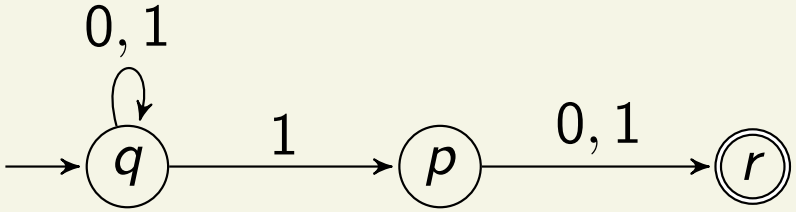
$B = \{x11 \mid x \in \{0,1\}^*\}$



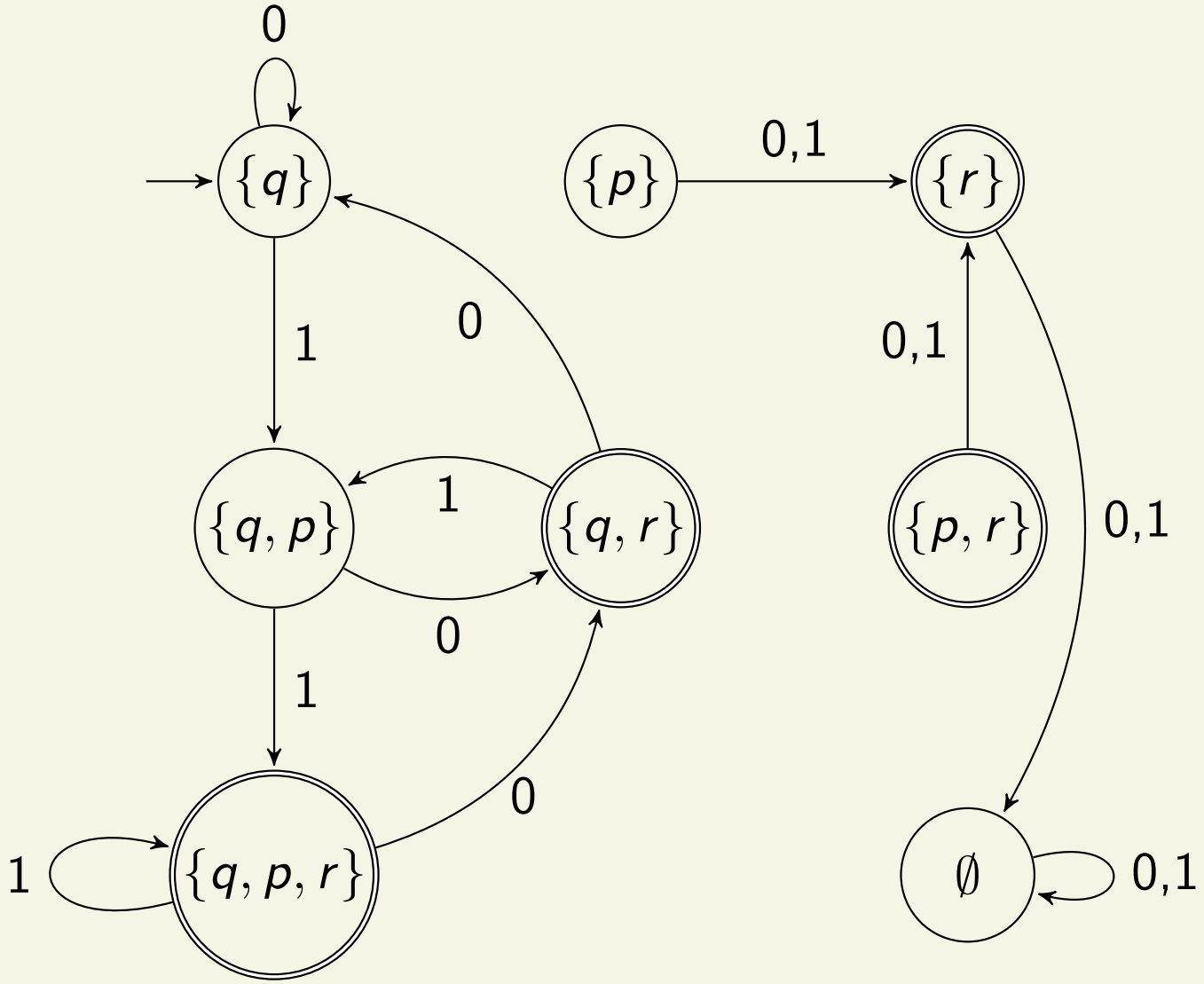
$A \cup B$:



NFA:

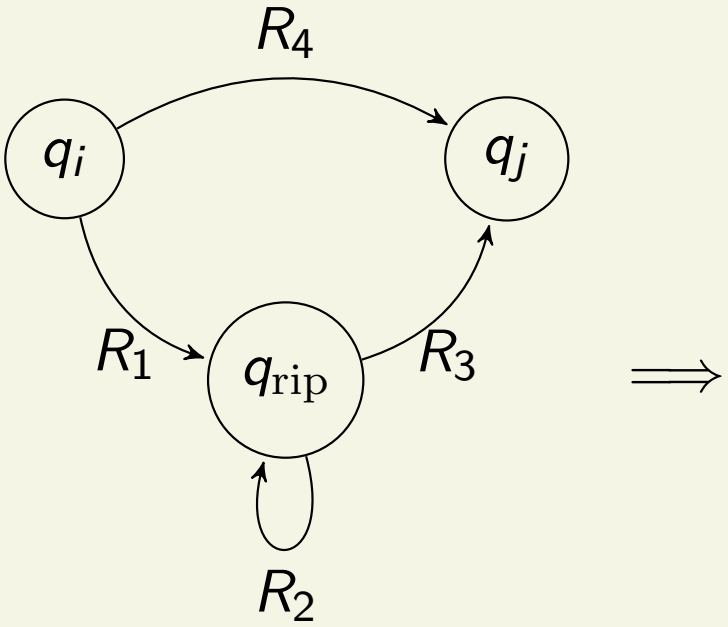


Subset Construction:



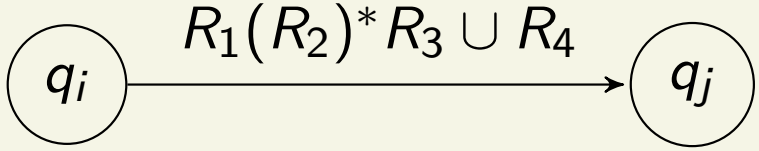
Converting a DFA to a Regular Expression

Select a state q_{rip} other than s or f to remove. For every pair of states q_i, q_j where there is a transition from q_i to q_{rip} and a transition from q_{rip} to q_j, do the following.



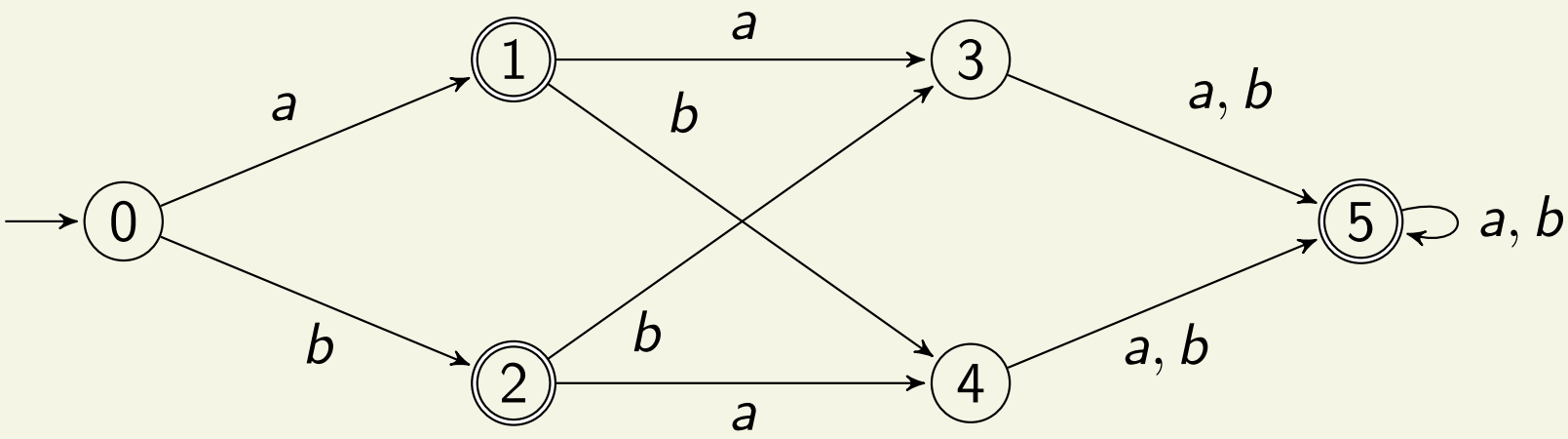
$\Longrightarrow$

New transition from q_i to q_j after q_{rip} is removed:



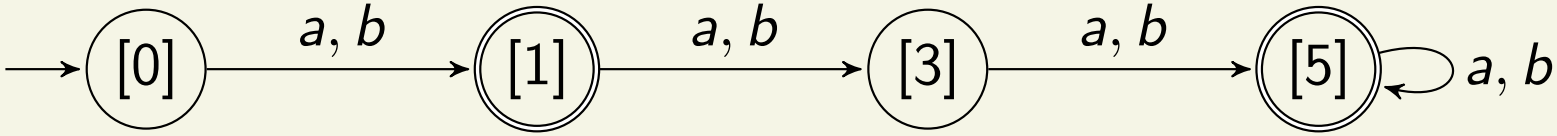
DFA Minimization and Quotient Automaton

Original DFA:



$$[0] = \{0\},\ [1] = \{1, 2\} = [2],\ [3] = \{3, 4\} = [4],\ [5] = \{5\}$$

Quotient Automaton:



- $\delta'([0], a) = [\delta(0, a)] = [1]$
- $\delta'([0], b) = [\delta(0, b)] = [2] = [1]$
- $\delta'([1], b) = [\delta(1, b)] = [4] = [3]$

Regular Closure Properties

REG is closed under:

- union
- concatenation
- star
- intersection
- complement

Pumping Lemma

If A is a regular language, then there is a number p (the pumping constant) such that for every $w \in A$ with $|w| \geq p$, w can be divided into three pieces $w = xyz$ satisfying the following conditions:

1. For all $i \geq 0$, $xy^i z \in A$.
2. $|y| > 0$.
3. $|xy| \leq p$.

Definition. Let $A \subseteq \Sigma^*$. An equivalence relation $\equiv$ on Σ^* is called a *Myhill-Nerode relation* for A if it is a right congruence of finite index that refines A.

Definition. Let $A \subseteq \Sigma^*$. We define an equivalence relation $\equiv_A$ on Σ^* by

$$x \equiv_A y \iff (\forall z \in \Sigma^*)\ (xz \in A \Leftrightarrow yz \in A)$$

for all $x, y \in \Sigma^*$.

Myhill-Nerode Theorem

Let $A \subseteq \Sigma^*$. The following are equivalent:

1. A is regular.
2. There exists a Myhill-Nerode relation for A.
3. The relation $\equiv_A$ is of finite index.

Examples of Nonregular Languages

- $\{0^n 1^n \mid n \geq 0\}$
- $\{w \in \{0,1\}^* \mid w \text{ is a palindrome}\}$
- $\{0^{n^2} \mid n \geq 0\}$
- $\{w \in \{0,1\}^* \mid \#_0(w) = \#_1(w)\}$
- $\{ww \mid w \in \{0,1\}^*\}$

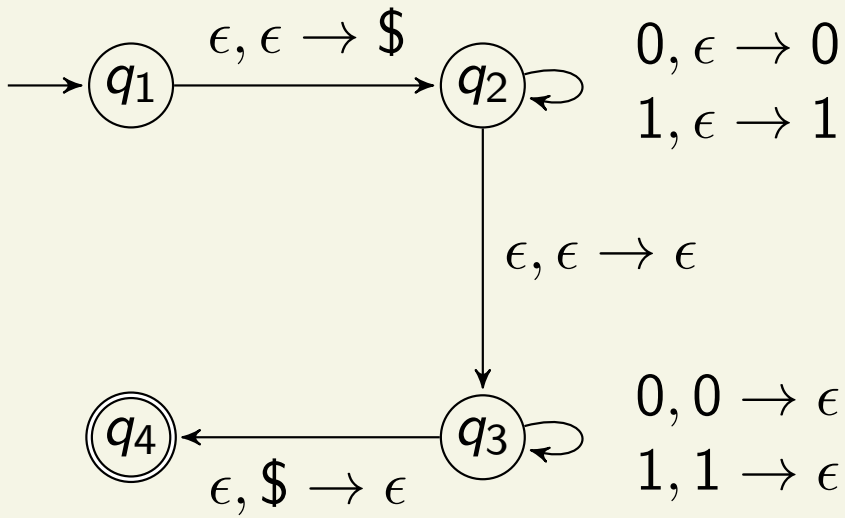
Pushdown Automata and Context-Free Languages

Context-Free Languages

Context-Free Grammars (CFG)

$$
\begin{array}{rcl}
S & \to & LC \mid AR \\
L & \to & aLb \mid \epsilon \\
R & \to & bRc \mid \epsilon \\
A & \to & Aa \mid \epsilon \\
C & \to & Cc \mid \epsilon
\end{array}
$$

Pushdown Automata (PDA)



Definition

A *context-free grammar (CFG)* is a 4-tuple $G = (V, \Sigma, R, S)$ where

- V is a finite set called the *variables* (usually V consists of capital letters)
- Σ is a finite set, disjoint from V, called the *terminals*
- R is a finite set of *rules*, with each rule begin of the form

$$R \to \omega$$

  where $R \in V$ and $\omega \in (V \cup \Sigma)^*$.
  (I.e., R is a variable and ω is a string of variables and terminals.)
- S is the *start variable*.

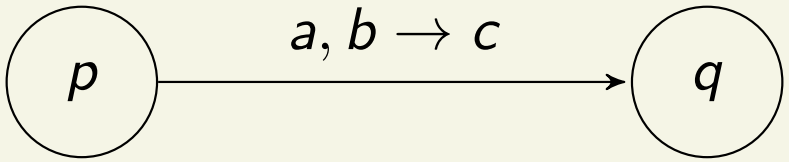
Definition

A *(nondeterministic) pushdown automaton* (*PDA*) is a 6-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ where

- Q is a finite set of *states*
- Σ is the *input alphabet*
- Γ is the *stack alphabet*
- $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to \mathcal{P}(Q \times \Gamma_\epsilon)$ is the *transition function*
- $q_0 \in Q$ is the *initial state*
- $F \subseteq Q$ is the set of *accepting states*

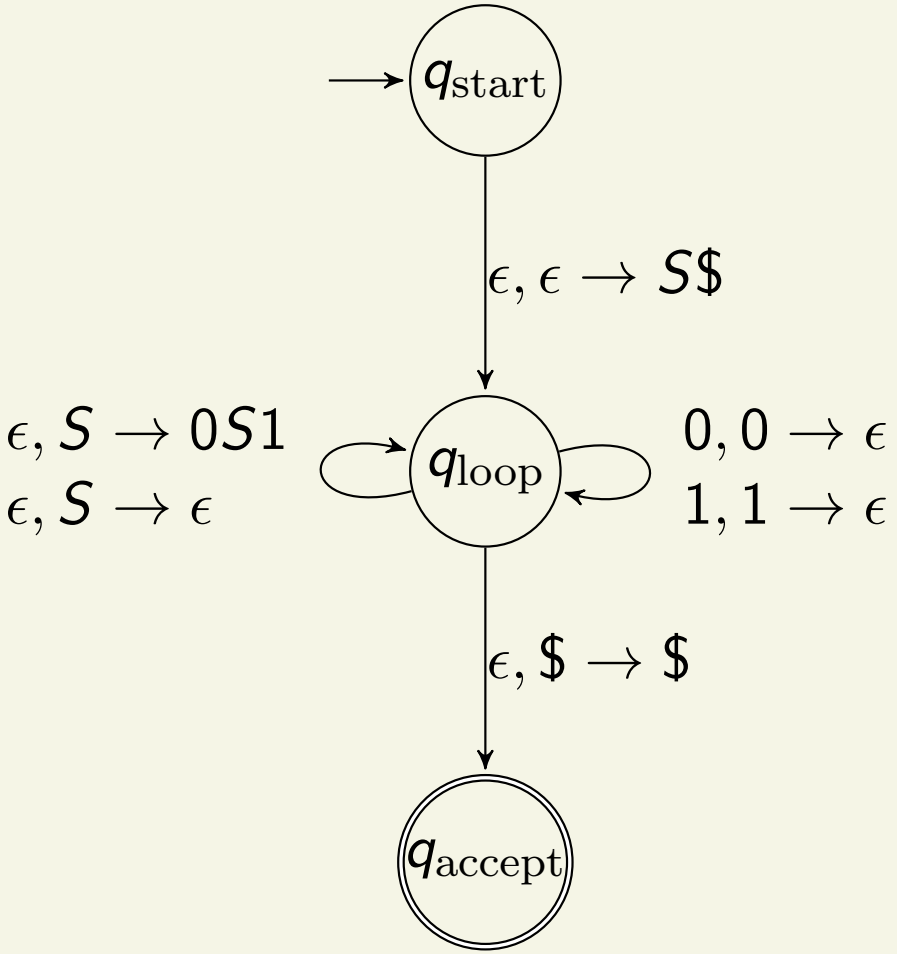
For each $p \in Q$, $a \in \Sigma_\epsilon$, $b \in \Gamma_\epsilon$, $\delta(p, a, b)$ is a subset of $Q \times \Gamma_\epsilon$.

For each $(q, c) \in \delta(p, a, b)$, we have the following transition in a diagram for the PDA:



Converting a CFG into a PDA

Consider the grammar $S \to 0S1 \mid \epsilon$.
The idea is to build the following PDA:

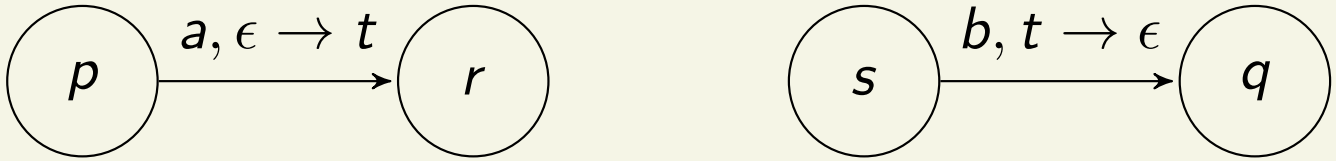


On q_{loop}, we have

- $a, a \to \epsilon$ for each $a \in \Sigma$
- $\epsilon, A \to \omega$ for each rule $A \to \omega$.

Converting a PDA to a CFG

1. For each $p, q, r, s \in Q$, $t \in \Gamma$, and $a, b \in \Sigma_\epsilon$, if $(r, t) \in \delta(p, a, \epsilon)$ and $(q, \epsilon) \in \delta(s, b, t)$,



   we have the rule

$$A_{p,q} \to aA_{r,s}b.$$

2. For each $p, q, r \in Q$, we have the rule

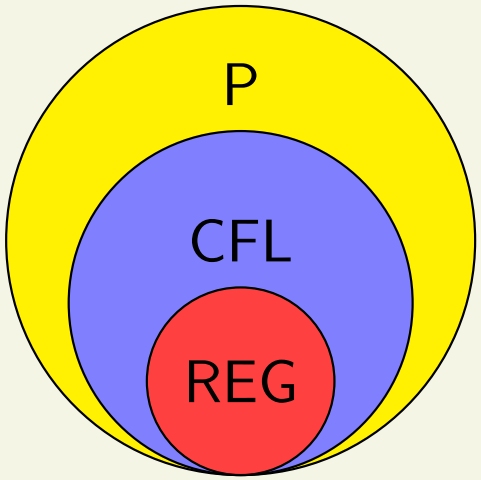
$$A_{p,q} \to A_{p,r}A_{r,q}.$$

3. For each $p \in Q$, we have the rule

$$A_{p,p} \to \epsilon.$$

Complexity of Context-Free Languages

- Every CFG may be converted into Chomsky normal form.
- Membership in a Chomsky normal form grammar may be decided in $O(n^3)$ time by the CKY algorithm.
- $\mathrm{REG} \subseteq \mathrm{CFL} \subseteq \mathrm{P}$.



Pumping Lemma for Context-Free Languages

If A is a context-free language, then there is a number p (the pumping constant) such that for every $w \in A$ with $|w| \geq p$, w can be divided into five pieces $w = uvxyz$ satisfying the following conditions:

1. For all $i \geq 0$, $uv^i xy^i z \in A$.
2. $|vy| > 0$.
3. $|vxy| \leq p$.

Summary of CFL Closure Properties

CFL is closed under:

- union
- concatenation
- star

CFL is not closed under:

- intersection
- complement

Examples of Non-Context-Free Languages

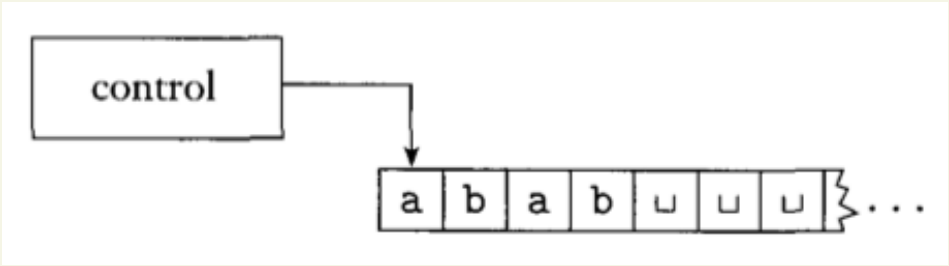
- $\{a^n b^n c^n \mid n \geq 0\}$
- $\{0^{n^2} \mid n \geq 0\}$
- $\{ww \mid w \in \{0,1\}^*\}$

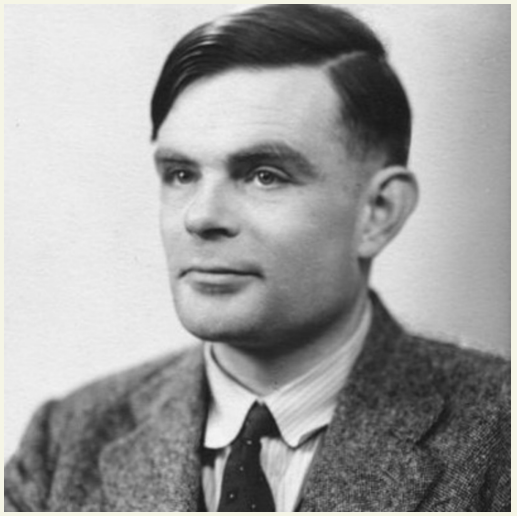
Turing Machines and Computability Theory

Turing Machines (Alan Turing, 1936)

Turing Machines

- simple model of a general purpose computer
- finite automaton with infinite read/write tape





Alan Turing
(1912-1954)

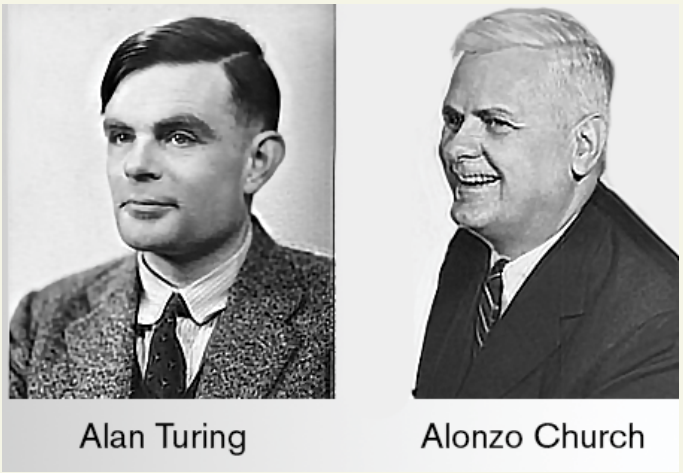
Church-Turing Thesis

Church-Turing Thesis

intuitive notion of algorithms

equals

Turing machine algorithms



Alan Turing

Alonzo Church

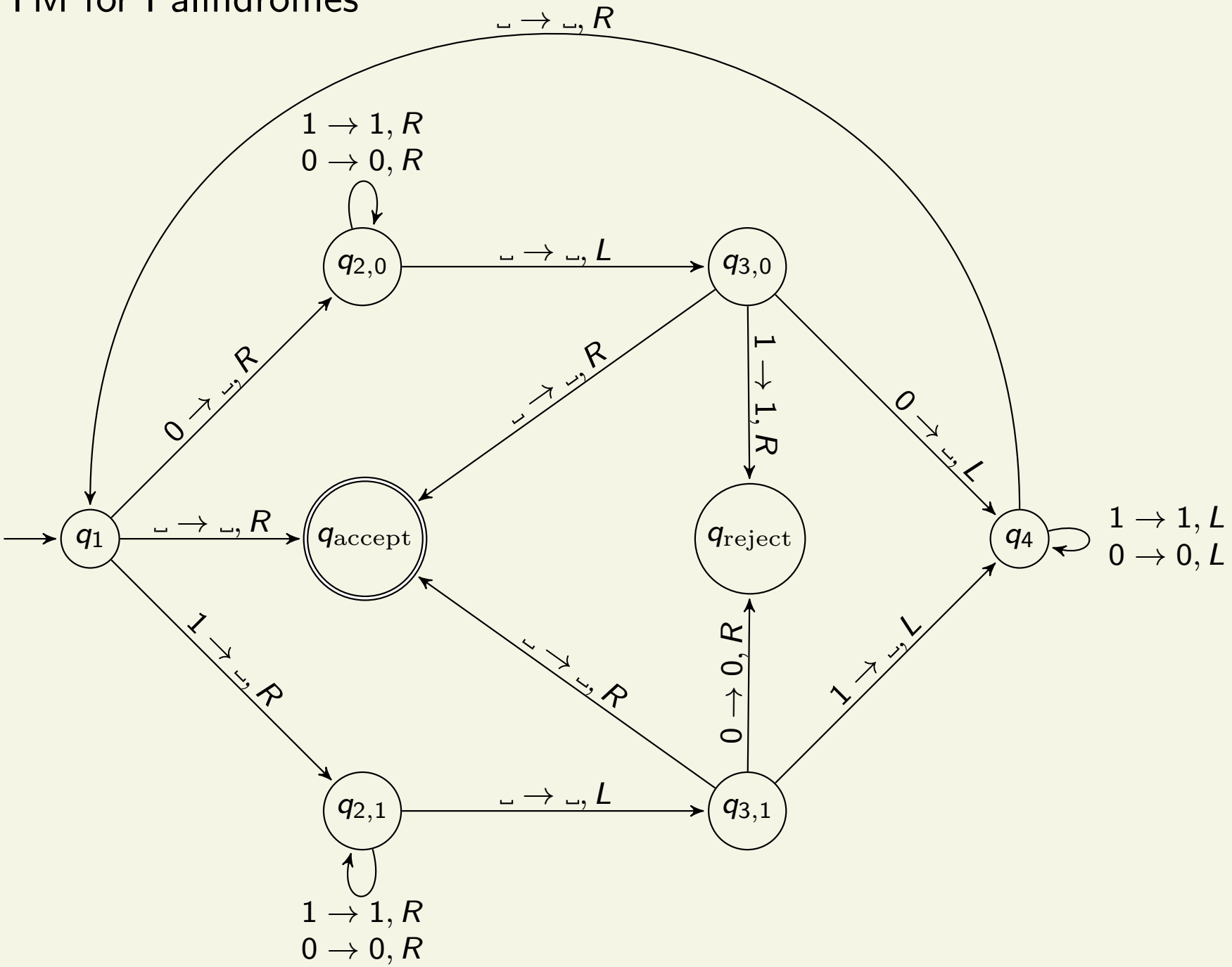
Formal Definition of a Turing Machine

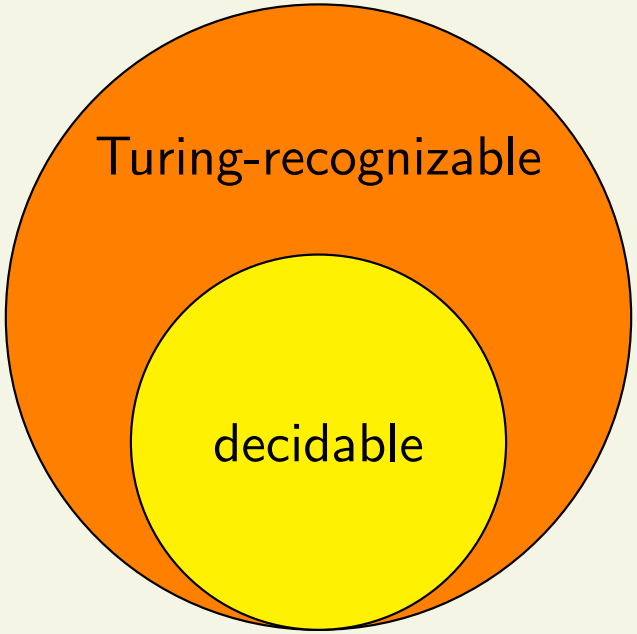
Definition

A *Turing machine* (*TM*) is a 7-tuple
$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ where

1. Q is a finite set of *states*
2. Σ is the *input alphabet* not containing the special *blank* symbol $\sqcup$
3. Γ is the *tape alphabet*, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the *transition function*
5. $q_0 \in Q$ is the *initial state*
6. $q_{\text{accept}} \in Q$ is the *accept state*
7. $q_{\text{reject}} \in Q$ is the *reject state*

TM for Palindromes





- A is *Turing-recognizable* if there is a TM M such that for all x,

$$\begin{aligned} x \in A &\Rightarrow M \text{ accepts } x \\ x \notin A &\Rightarrow M \text{ does not accept } x \\ &\quad (M \text{ rejects } x \text{ or } M \text{ does not halt on } x) \end{aligned}$$

- A is *decidable* if there is a TM M such that for all x,

$$\begin{aligned} x \in A &\Rightarrow M \text{ accepts } x \\ x \notin A &\Rightarrow M \text{ rejects } x \end{aligned}$$

  In other words, A is recognized by an always-halting TM.

A is *decidable* if there is a TM M such that for all w,

- $w \in A \Rightarrow M$ accepts w.
- $w \notin A \Rightarrow M$ rejects w.

A is *Turing-recognizable* if there is a TM M such that for all w,

- $w \in A \Rightarrow M$ accepts w.
- $w \notin A \Rightarrow M$ does not accept w.

A is *co-Turing-recognizable* if there is a TM M such that for all w,

- $w \in A \Rightarrow M$ does not accept w.
- $w \notin A \Rightarrow M$ accepts w.

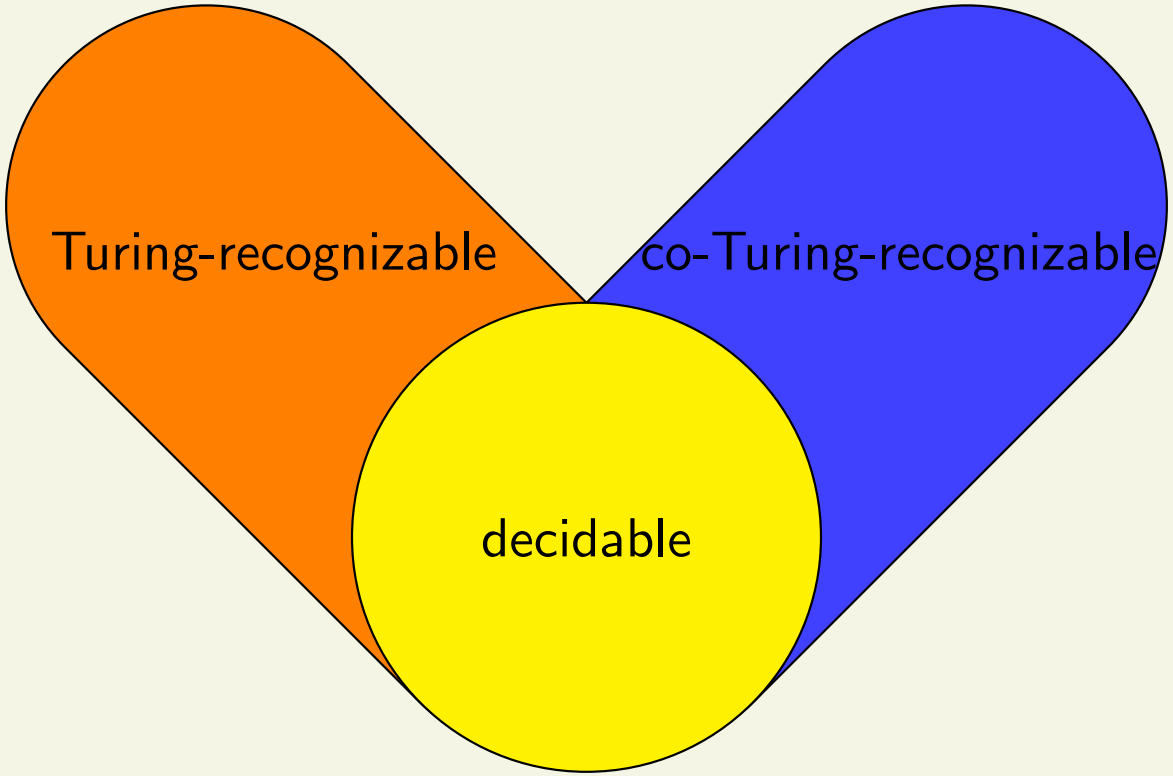
Theorem

A language is decidable if and only if both it is both Turing-recognizable and co-Turing-recognizable.

Decidable = Turing-recognizable ∩ co-Turing-recognizable

Theorem

A language is decidable if and only if both it is both Turing-recognizable and co-Turing-recognizable.



Mapping Reducibility

Definition

A function $f : \Sigma^* \to \Sigma^*$ is a *computable function* if some Turing machine M, on every input w halts with just $f(w)$ on its tape.

Definition

Language A is *mapping reducible* to language B, written $A \leq_{\mathrm{m}} B$, if there is a computable function $f : \Sigma^* \to \Sigma^*$, where for every w,

$$w \in A \Leftrightarrow f(w) \in B.$$

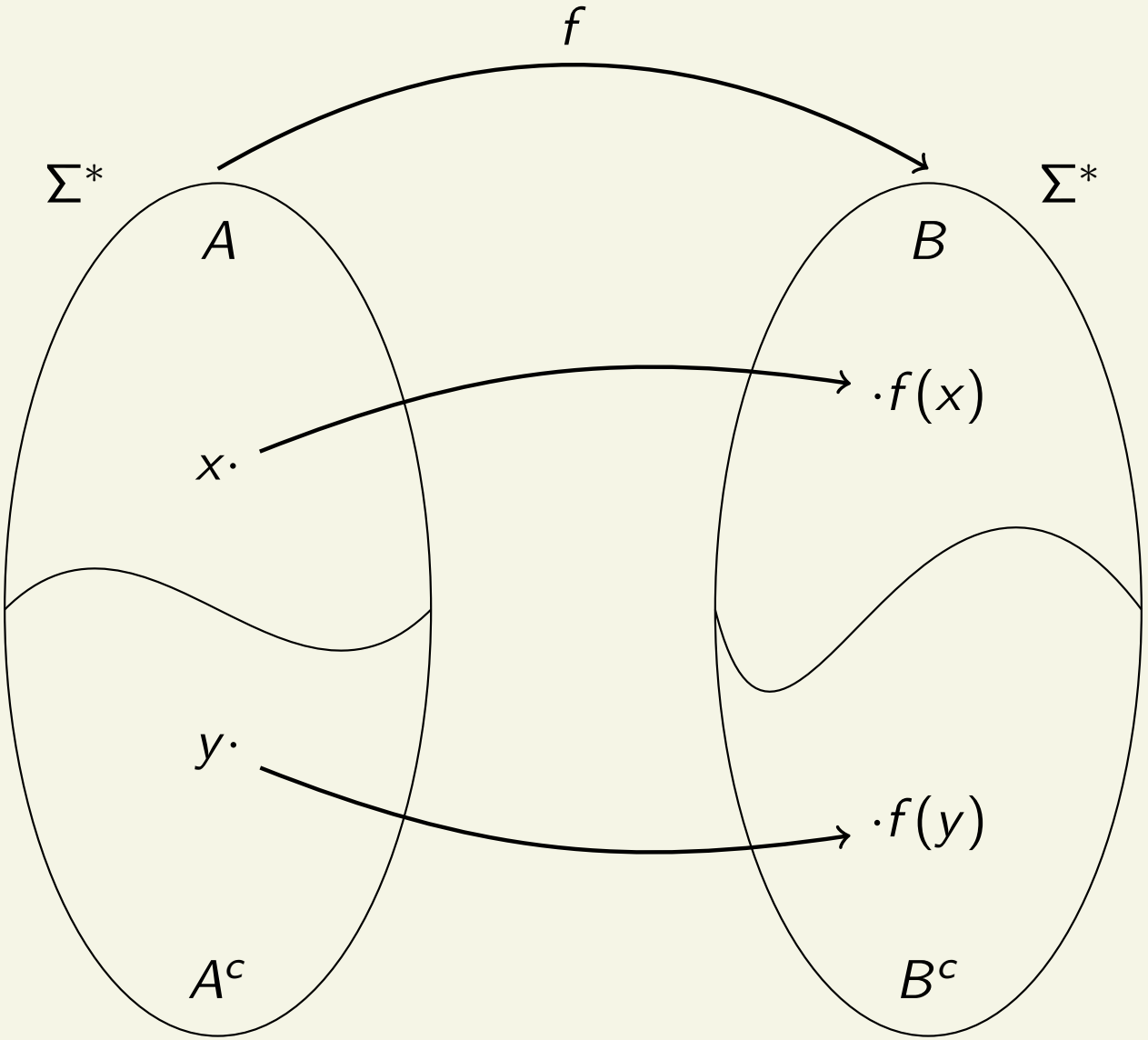
The function f is called a *reduction* of A to B.

The reduction maps positive instances to positive instances:

$$w \in A \Rightarrow f(w) \in B$$

and negative instance to negative instances:

$$w \notin A \Rightarrow f(w) \notin B.$$



$A \leq_{\mathrm{m}} B$ via f

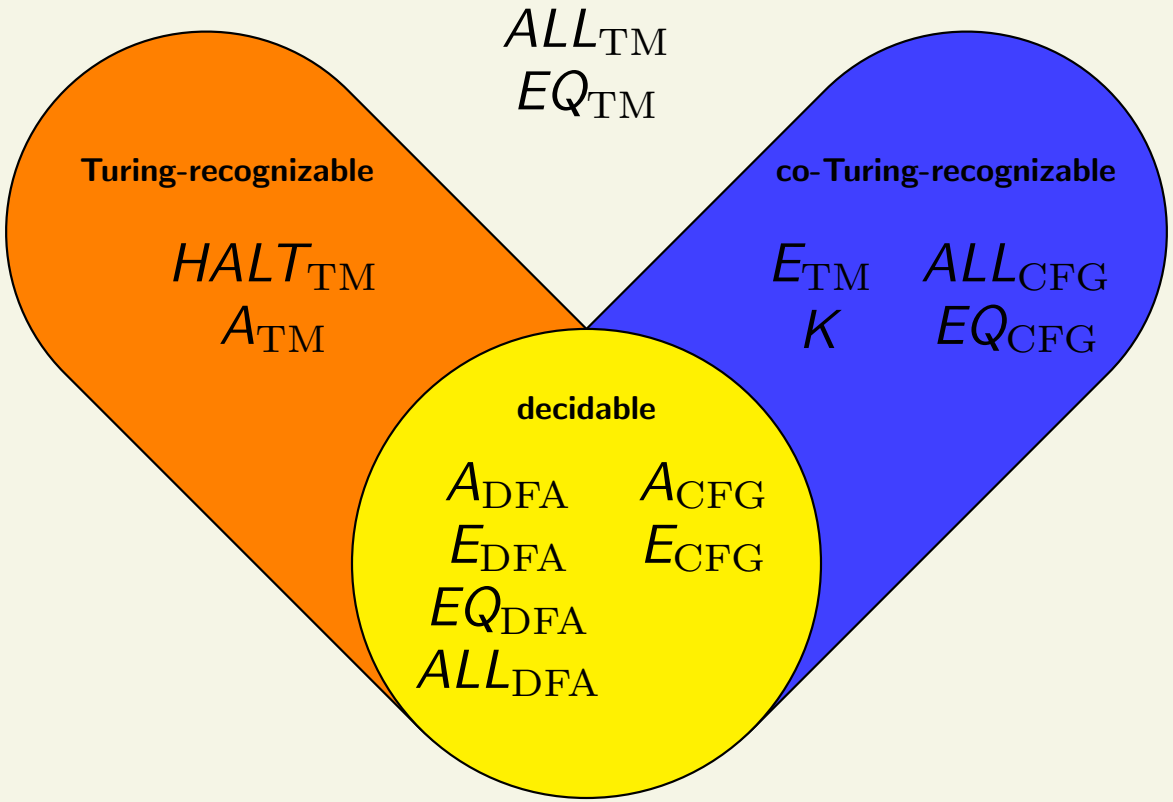
Theorem

If $A \leq_{\mathrm{m}} B$ and B is decidable, then A is decidable.

This provides a tool for proving undecidability:

Corollary

If $A \leq_{\mathrm{m}} B$ and A is undecidable, then B is undecidable.



	DFA	CFG	TM
A (acceptance)	decidable	decidable	Turing-recognizable not co-Turing-recognizable
E (emptiness)	decidable	decidable	co-Turing-recognizable not Turing-recognizable
ALL (all strings)	decidable	co-Turing-recognizable not Turing-recognizable	not Turing-recognizable not co-Turing-recognizable
EQ (equivalence)	decidable	co-Turing-recognizable not Turing-recognizable	not Turing-recognizable not co-Turing-recognizable

Rice's Theorem

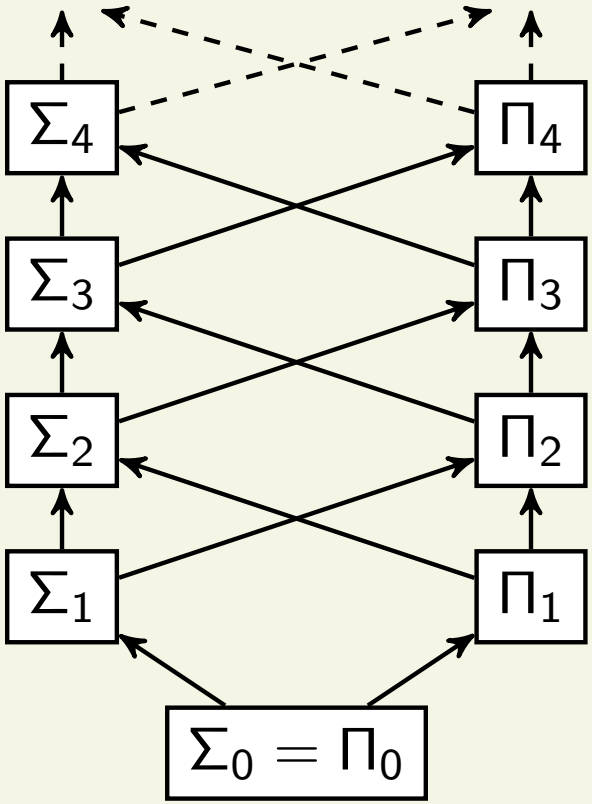
Let P be any problem about Turing machines that satisfies the following two properties. ($P \subseteq \{\langle M \rangle \mid M \text{ is a TM}\}$)

- For any two TMs M_1 and M_2, where $L(M_1) = L(M_2)$, we have $\langle M_1 \rangle \in P \Leftrightarrow \langle M_2 \rangle \in P$. In other words, the membership of $\langle M \rangle \in P$ depends only on $L(M)$.
- There exist two TMs M_1 and M_2, where $\langle M_1 \rangle \in P$ and $\langle M_2 \rangle \notin P$. In other words, P is nontrivial – it holds for some, but not all TMs.

Then P is undecidable.

Arithmetical Hierarchy

The Arithmetical Hierarchy is a collection of classes that sit above Turing-recognizable and co-Turing-recognizable.

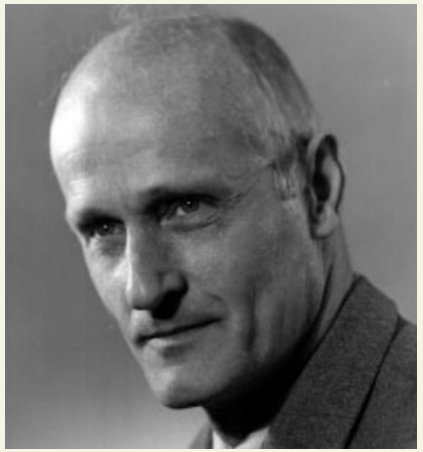


The Arithmetical Hierarchy

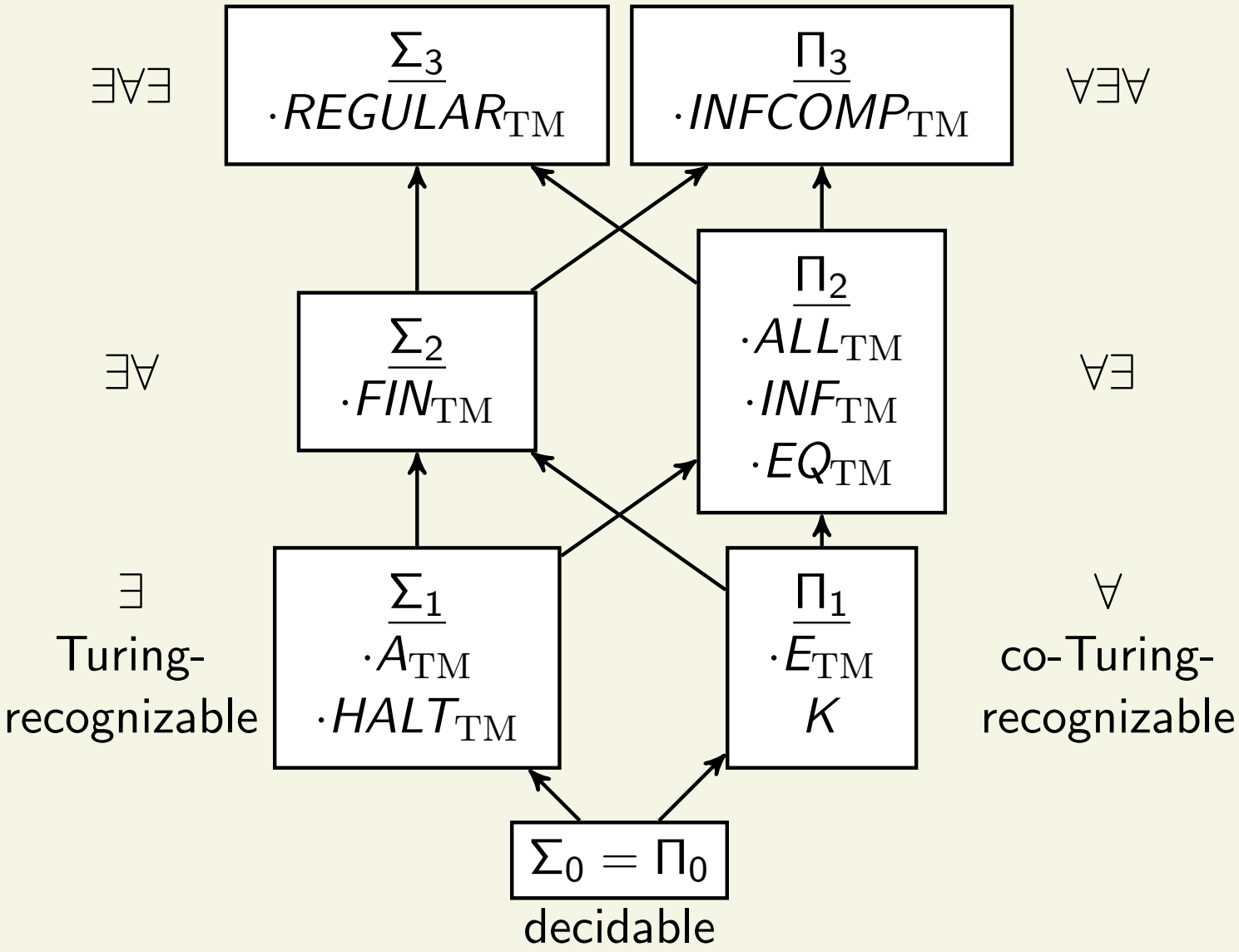
$\Pi_0 = \Sigma_0 =$ decidable

$\Sigma_1 =$ Turing-recognizable

$\Pi_1 =$ co-Turing-recognizable

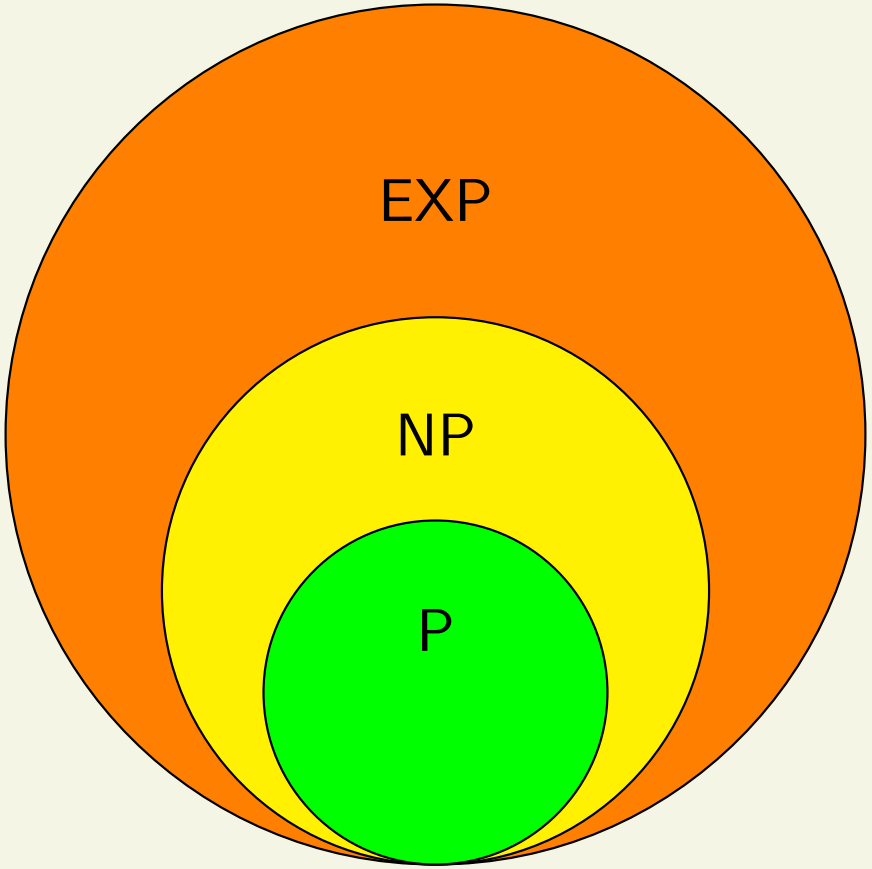


Stephan Kleene
(1909-1994)



The Arithmetical Hierarchy

Polynomial-Time Algorithms and Computational Complexity



$$\begin{aligned} \mathrm{P} &= \text{deterministic polynomial-time} \\ \mathrm{NP} &= \text{nondeterministic polynomial-time} \\ \mathrm{EXP} &= \text{deterministic exponential-time} \end{aligned}$$

Major open problems in computational complexity:

- Does P = NP?
- Does NP = EXP?

Polynomial-Time Church-Turing Thesis

There is an extension of the Church-Turing Thesis for polynomial-time computation:

Polynomial-Time Church-Turing Thesis

Let $\mathcal{M}$ be any *reasonable* deterministic computation model (such as random access machines or multitape TMs).

Then any language which can be decided in polynomial-time on model $\mathcal{M}$ can also be decided in polynomial-time by a single-tape TM, and vice versa.

Therefore the definition of P does not depend on the model of computation.

Verifiers and NP

Definition

A *verifier* for a language A is an algorithm V such that

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}.$$

c is called a *certificate* (or *witness*) that w belongs to A.

In other words:

$$\begin{aligned} w \in A &\Rightarrow (\exists c)\ V \text{ accepts } \langle w, c \rangle \\ w \notin A &\Rightarrow (\forall c)\ V \text{ rejects } \langle w, c \rangle \end{aligned}$$

Definition

NP is the class of languages that have polynomial-time verifiers.

Technical note: the verifier run time must be polynomial in $|w|$.

Verifiers versus Nondeterminism

The original definition of NP used polynomial-time nondeterministic Turing machines.

The definition using polynomial-time verifiers is equivalent.

Theorem

The following are equivalent for any language A.

1. *A is accepted by a polynomial-time NTM.*
2. *A has a polynomial-time verifier.*

Quantifiers and Predicates

We can rephrase the verifier definition of NP using a polynomial-time predicate with polynomial-size witnesses:

Theorem

$A \in \mathrm{NP}$ if and only if there is a $D \in \mathrm{P}$ and a polynomial p such that for all $x \in \Sigma^*$,

$$x \in A \iff (\exists w \in \{0,1\}^{\leq p(n)})\ \langle x, w \rangle \in D.$$

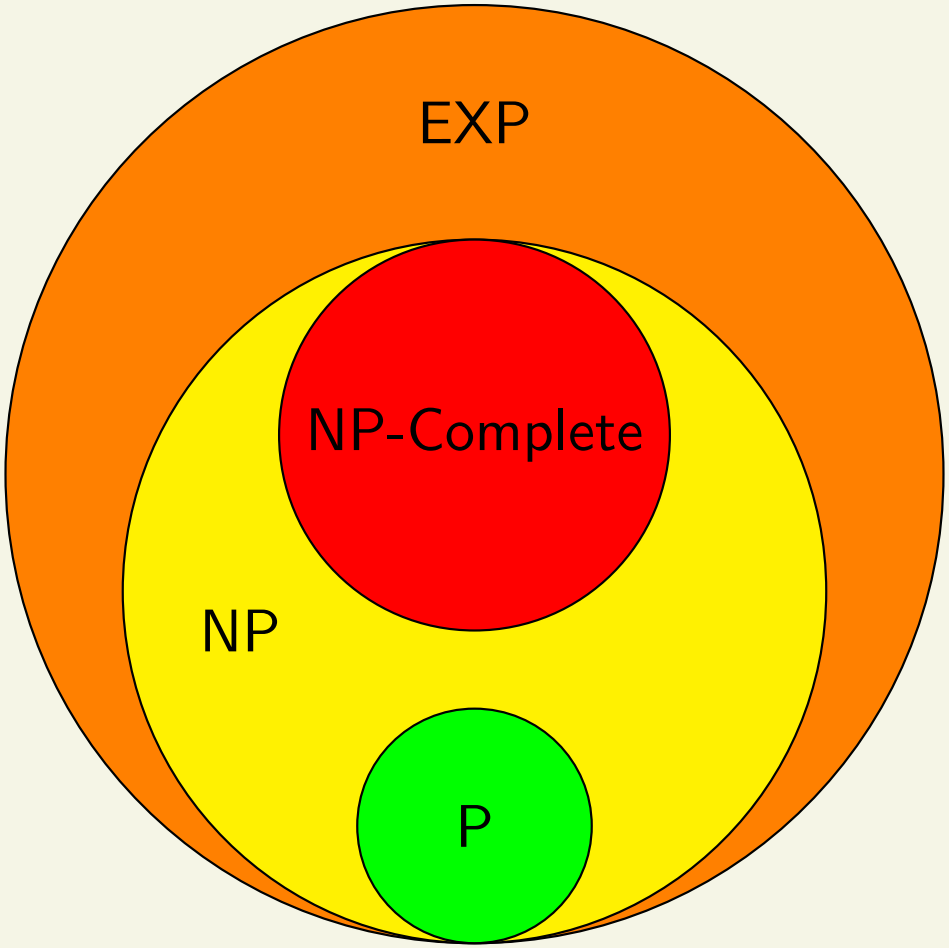
This is analogous to the Σ_1 definition of Turing-recognizable.

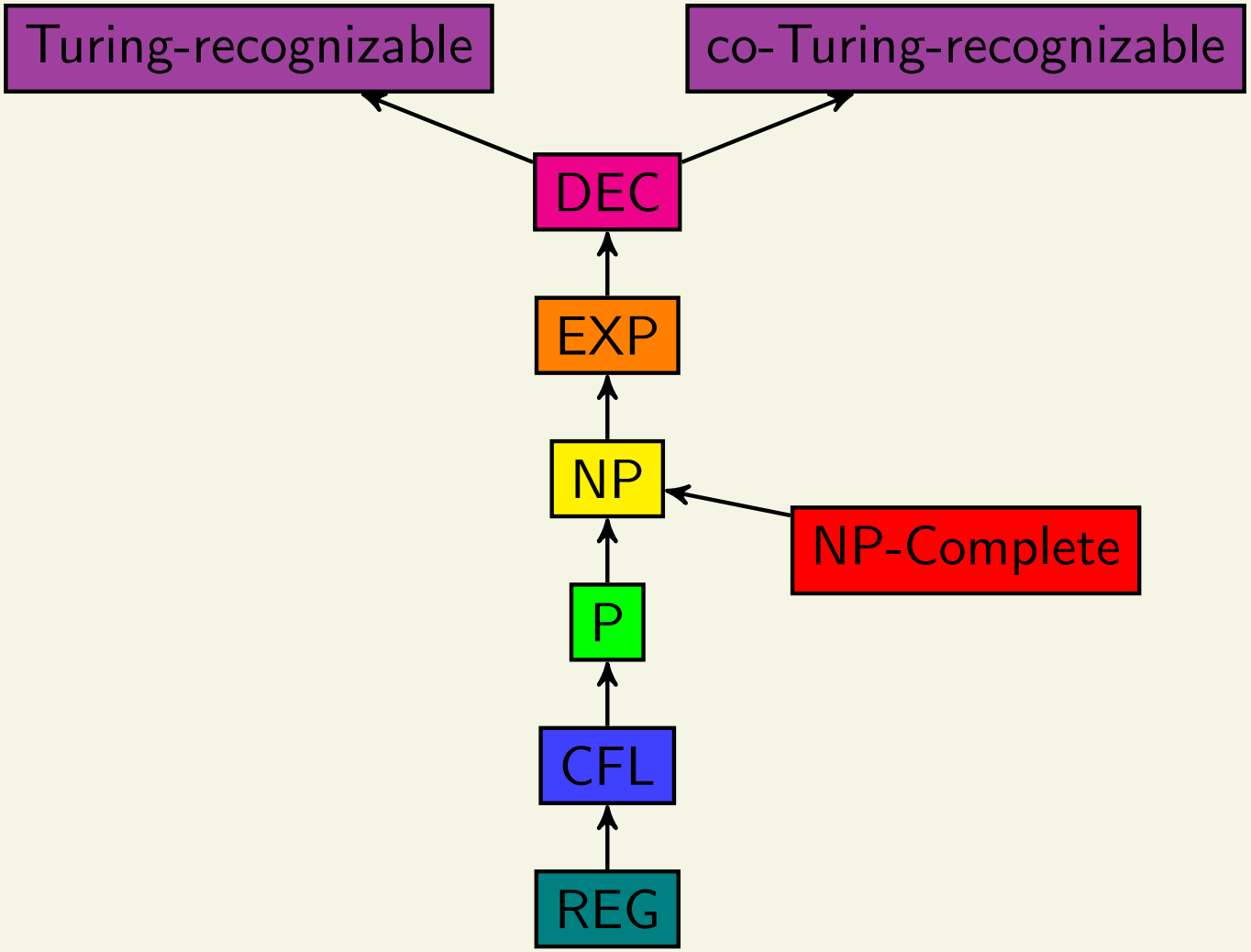
NP-Completeness

Definition

A language B is *NP-complete* if

1. B is in NP, and
2. every A in NP is polynomial-time reducible to B.


EXP
NP-Complete
NP
P


Turing-recognizable
co-Turing-recognizable
DEC
EXP
NP
NP-Complete
P
CFL
REG

NP-Completeness

Theorem

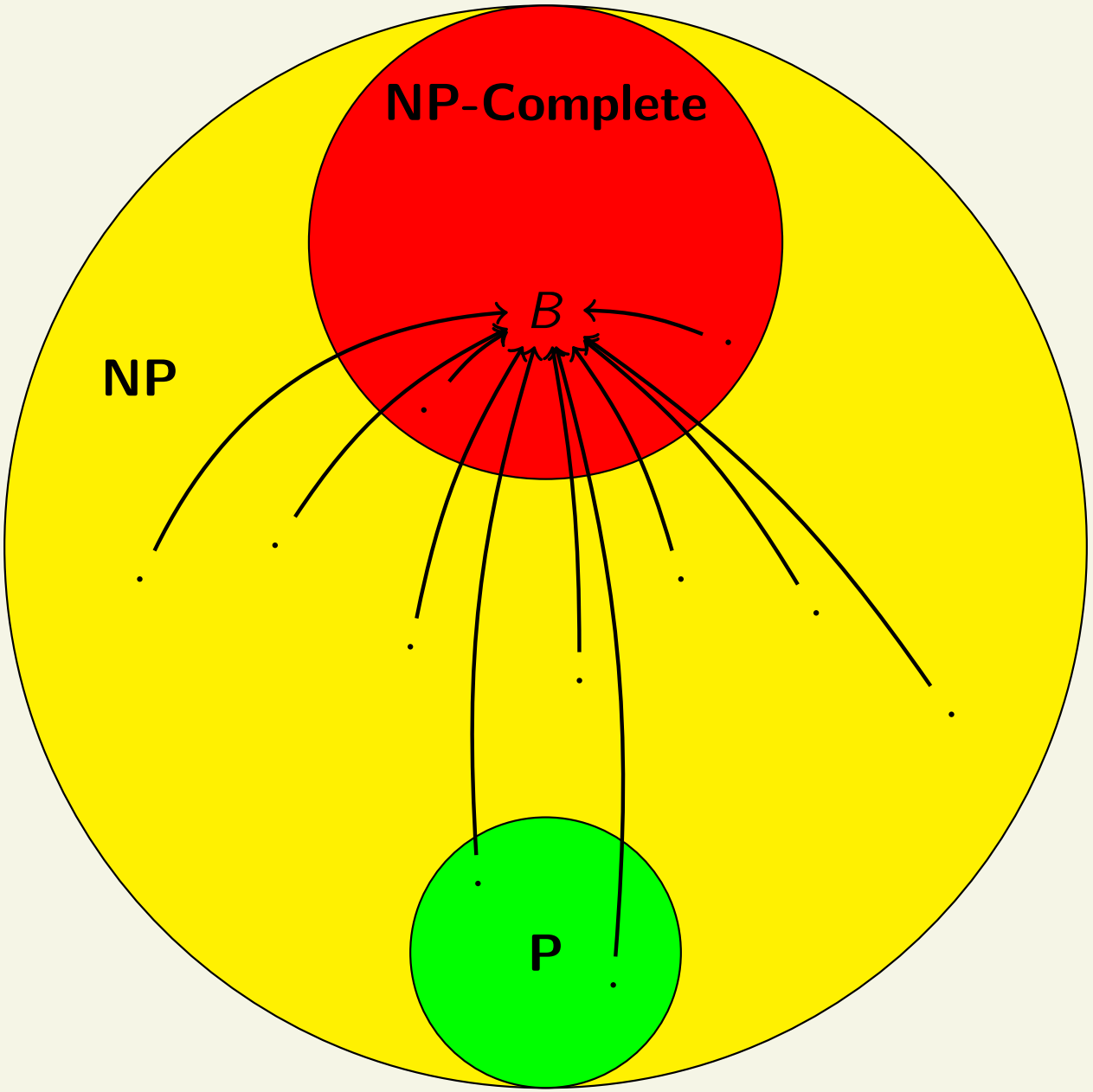
The following are equivalent:

1. *Some* NP*-complete problem is in* P*.*
2. *Every* NP*-complete problem is in* P*.*
3. $\mathrm{P} = \mathrm{NP}$*.*

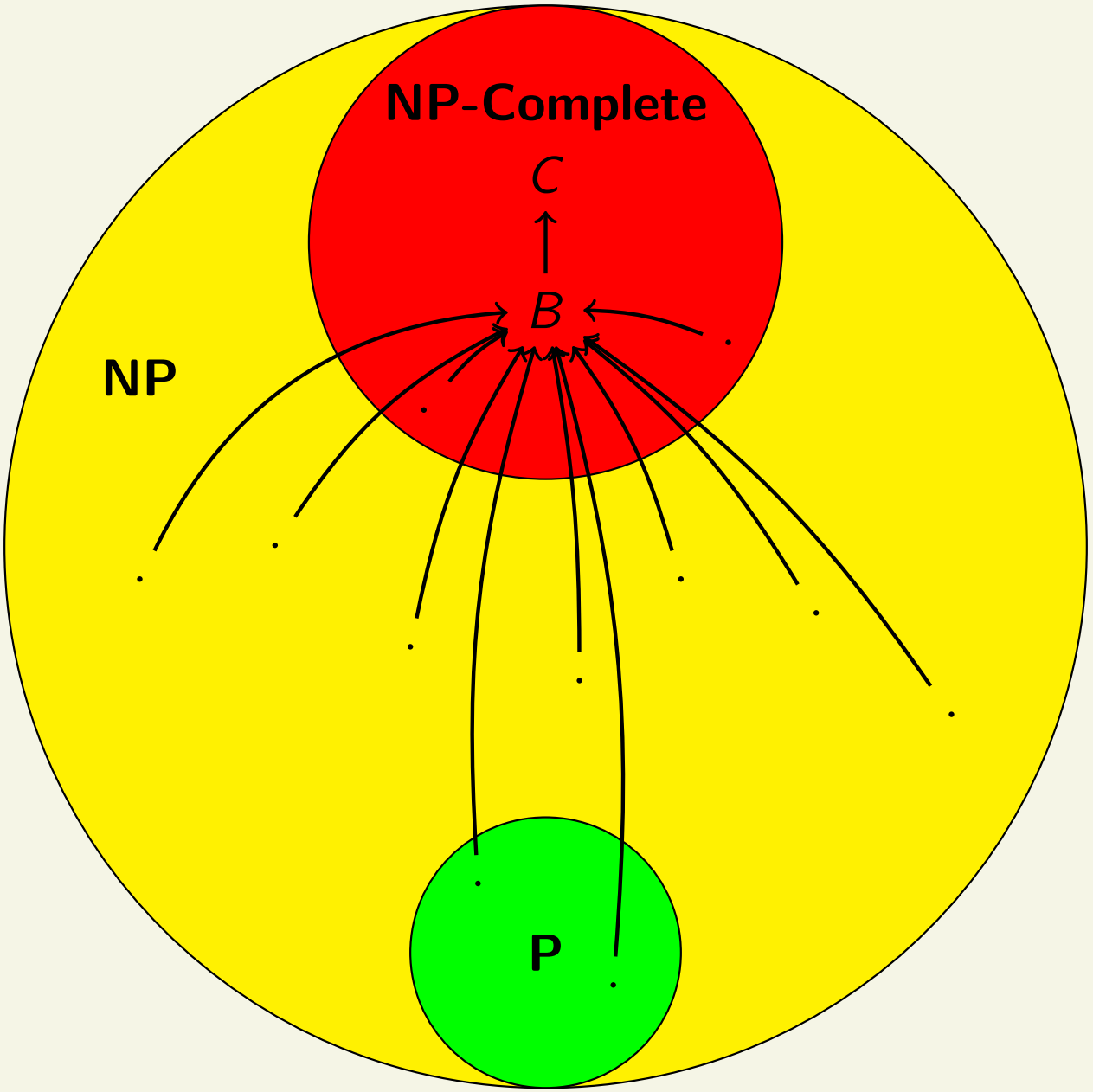
Corollary

The following are equivalent:

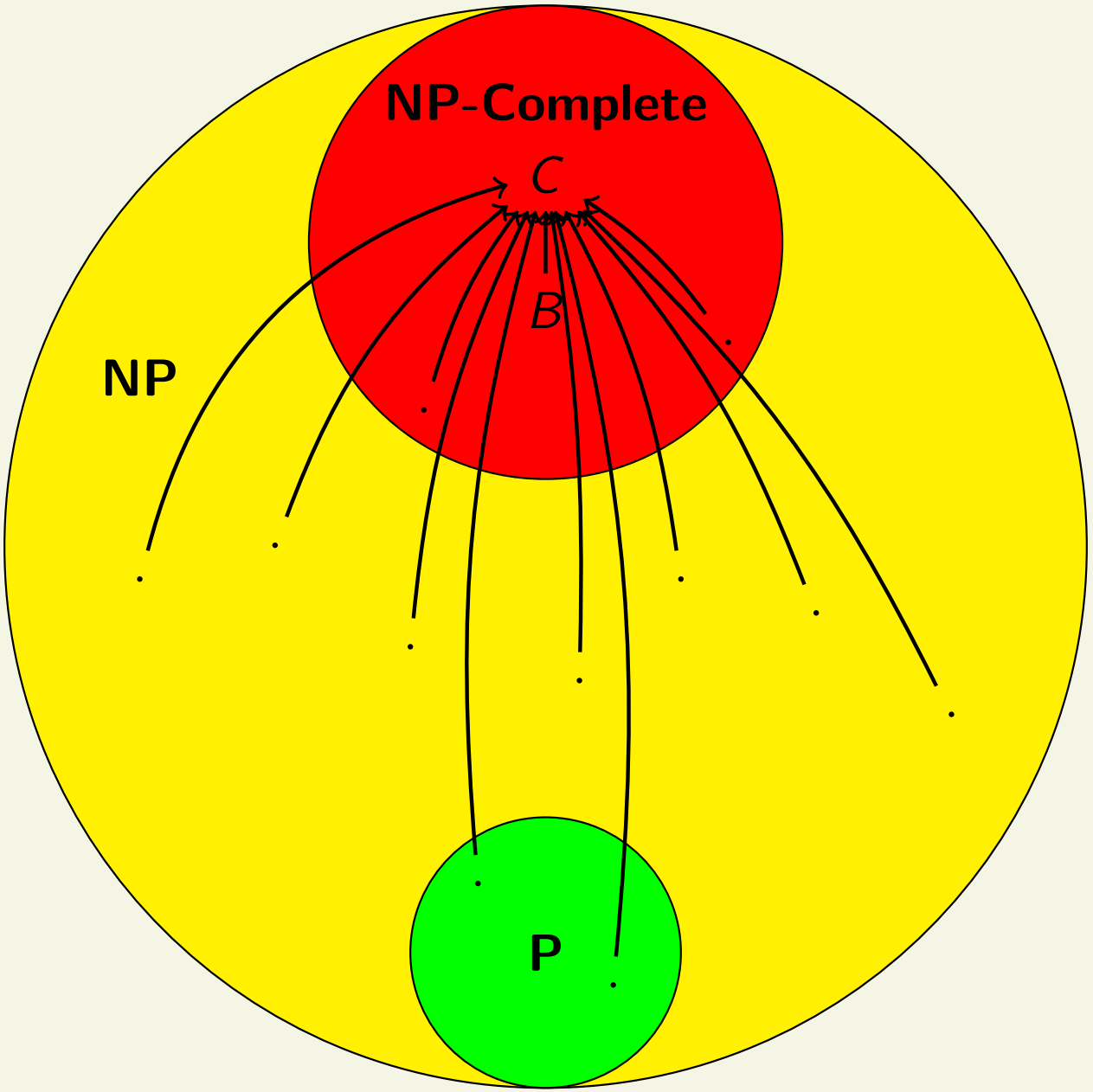
1. *Some* NP*-complete problem is not in* P*.*
2. *Every* NP*-complete problem is not in* P*.*
3. $\mathrm{P} \neq \mathrm{NP}$*.*



B is NP-complete

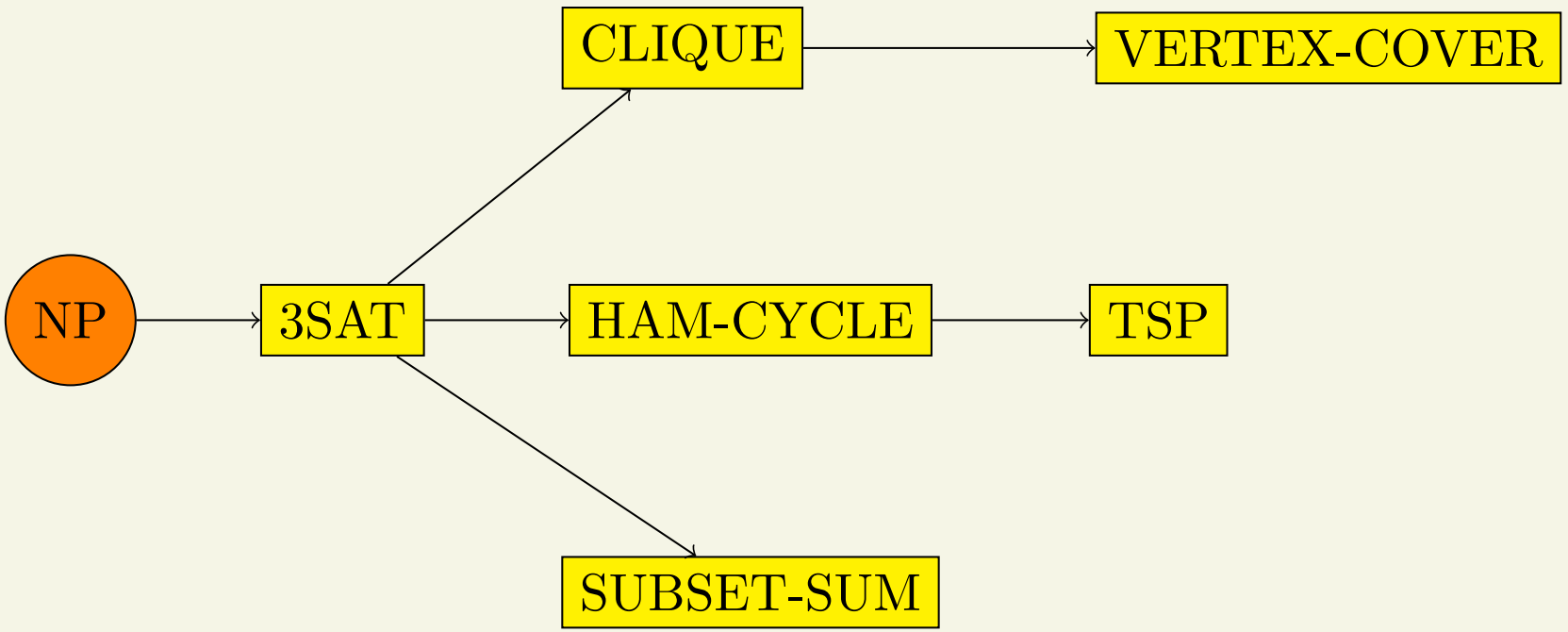


B is NP-complete and $B \leq_{\mathrm{P}} C \Rightarrow C$ is NP-complete



B is NP-complete and $B \leq_{\mathrm{P}} C \Rightarrow C$ is NP-complete

Summary of NP-completeness reductions:

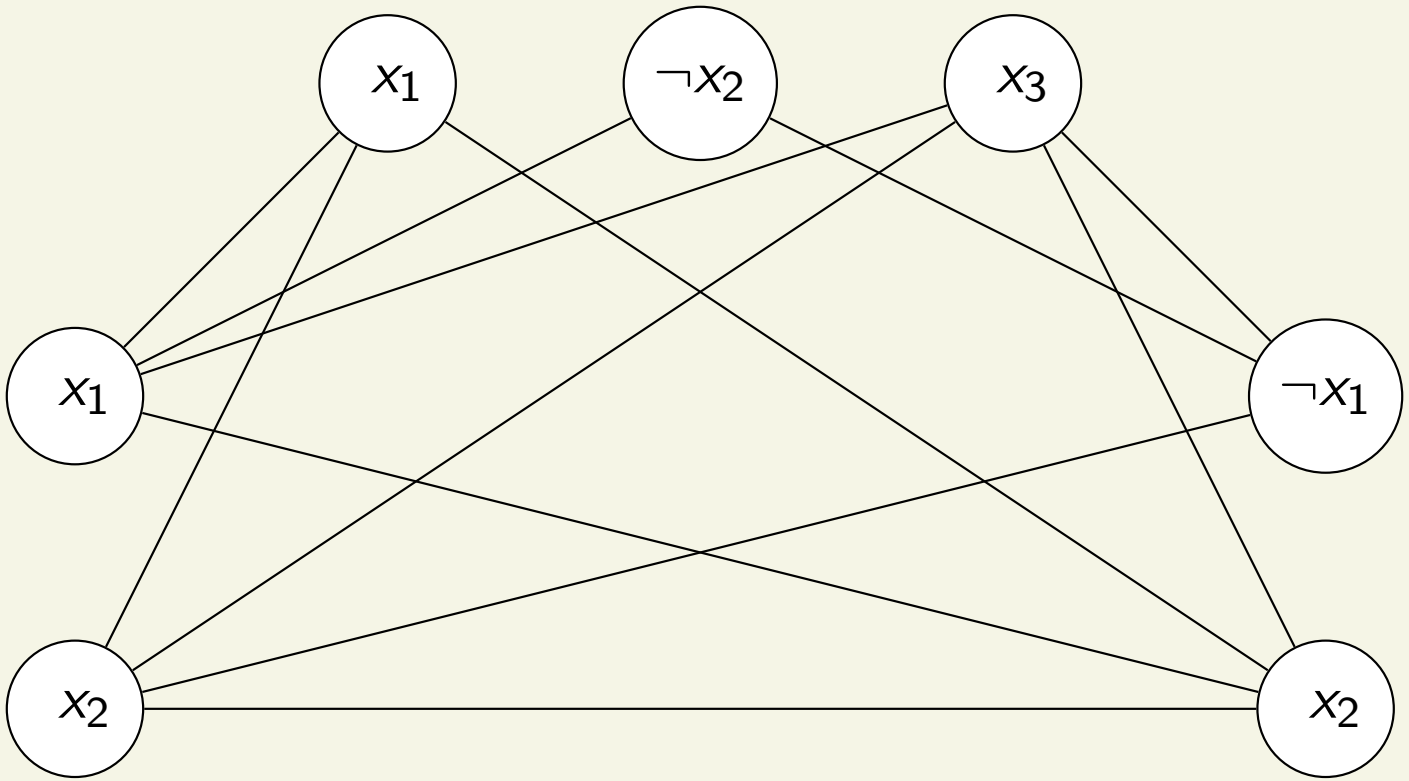


3SAT $\leq_{\mathrm{P}}$ CLIQUE

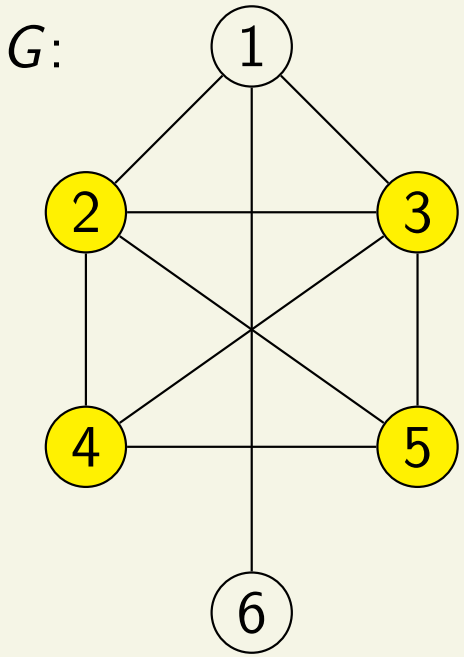
For example, consider the formula

$$\phi = (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2).$$

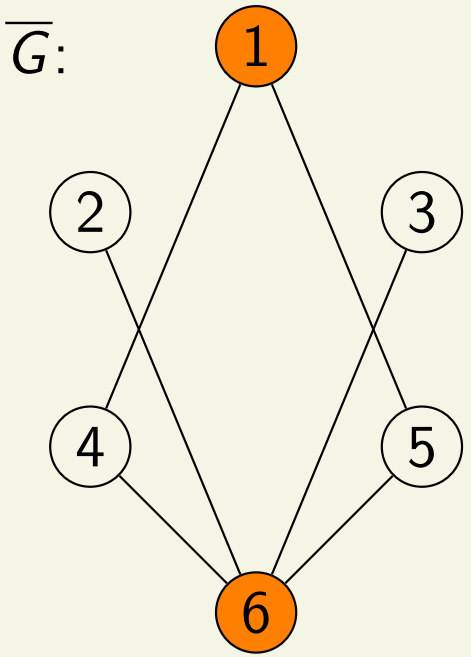
The graph G appears below. The three vertices at the top correspond to the first clause, the two at the left are for the second clause, and the two at the right are for the third clause.



CLIQUE $\leq_{\mathrm{P}}$ VERTEX-COVER

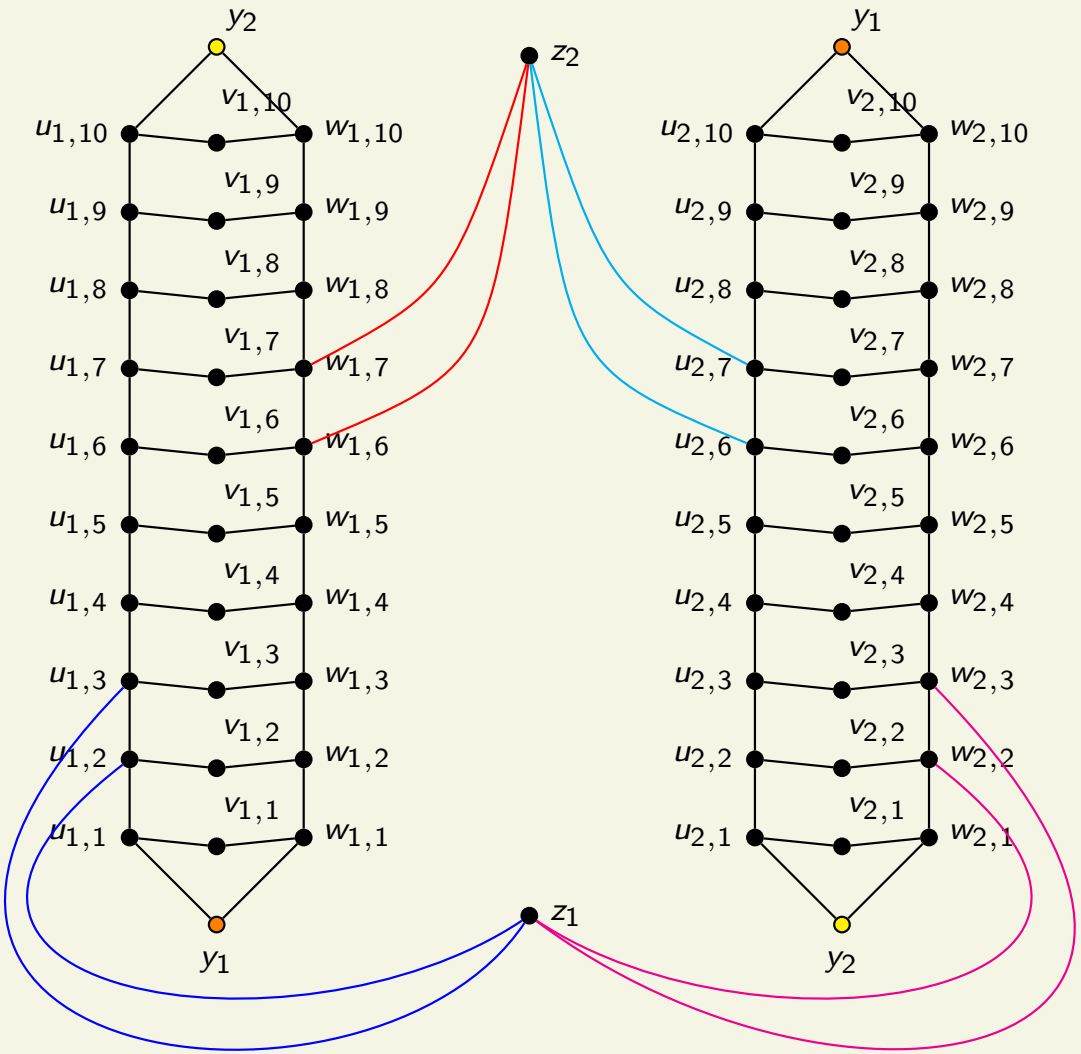


$V' = \{2, 3, 4, 5\}$
is a clique of size 4



$V - V' = \{1, 6\}$
is a vertex cover of size 2

3SAT $\leq_{\mathrm{P}}$ HAM-CYCLE



$$\phi = (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2)$$

For each clause C_j, we add a new vertex z_j such that

- If C_j contains the literal x_i, we connect z_j to $u_{i,4j-1}$, and $u_{i,4j-2}$.
- If C_j contains the literal $\neg x_i$, we connect z_j to $w_{i,4j-1}$ and $w_{i,4j-2}$.

HAM-CYCLE $\leq_{\mathrm{P}}$ TSP

Let $G = (V, E)$ be a graph.

Define $n = |V|$, $k = n$, and

$$c(i,j) = \begin{cases} 1 & \text{if } (i,j) \in E \\ n+1 & \text{if } (i,j) \notin E \end{cases}$$

for all $i, j \in \{1, \ldots, n\}$.

Then

$$G \in \mathrm{HC} \iff \langle n, c, k \rangle \in \mathrm{TSP}.$$

3SAT $\leq_{\mathrm{P}}$ SUBSET-SUM

As an example, consider

$$\phi = (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3).$$

Then $n = m = 3$, $S = 111333$, and

$$\begin{array}{lcl} b_{1,0} & = & 100101 \\ b_{1,1} & = & 100010 \\ b_{2,0} & = & 010011 \\ b_{2,1} & = & 010100 \\ b_{3,0} & = & 001101 \\ b_{3,1} & = & 001010 \end{array} \qquad \begin{array}{lcl} c_{1,0} & = & 000100 \\ c_{1,1} & = & 000100 \\ c_{2,0} & = & 000010 \\ c_{2,1} & = & 000010 \\ c_{3,0} & = & 000001 \\ c_{3,1} & = & 000001 \end{array}$$

Search Reduces to Decision

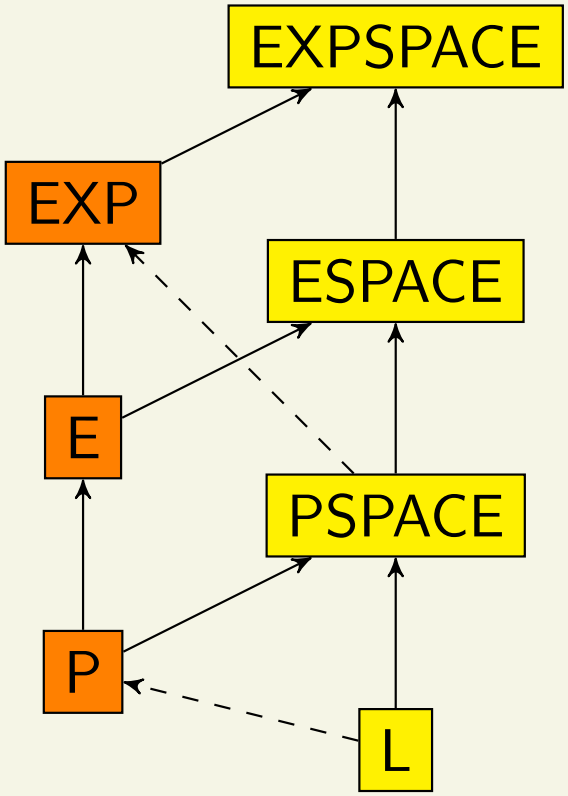
We know that SAT is NP-complete, so $\mathrm{SAT} \in \mathrm{P}$ if and only if $\mathrm{P} = \mathrm{NP}$. The following theorem shows that finding satisfying assignments is also equivalent to $\mathrm{P} = \mathrm{NP}$.

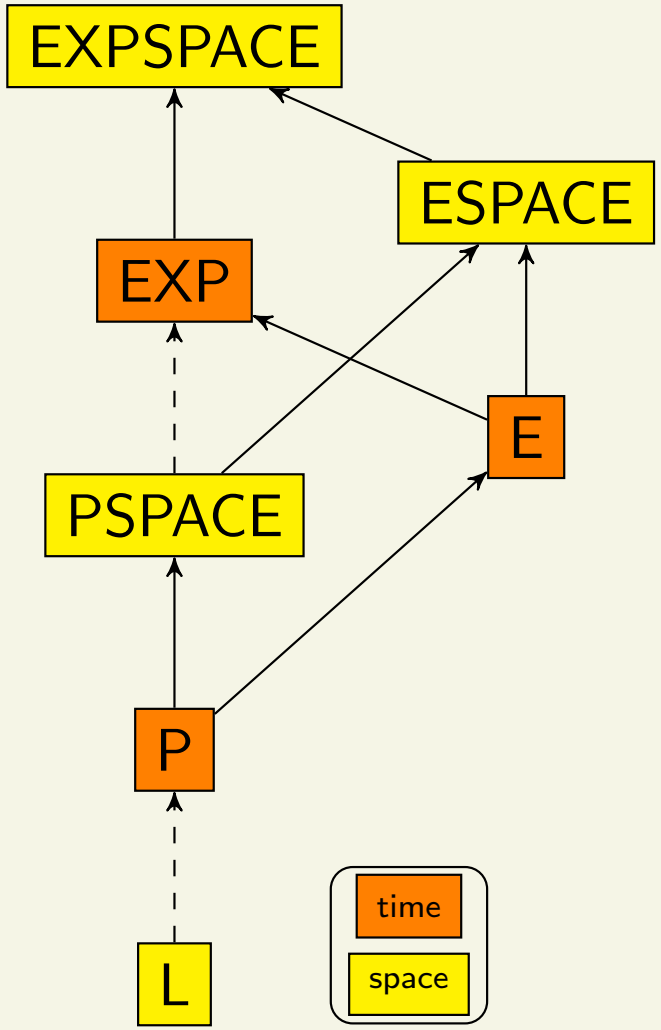
Theorem

The following are equivalent.

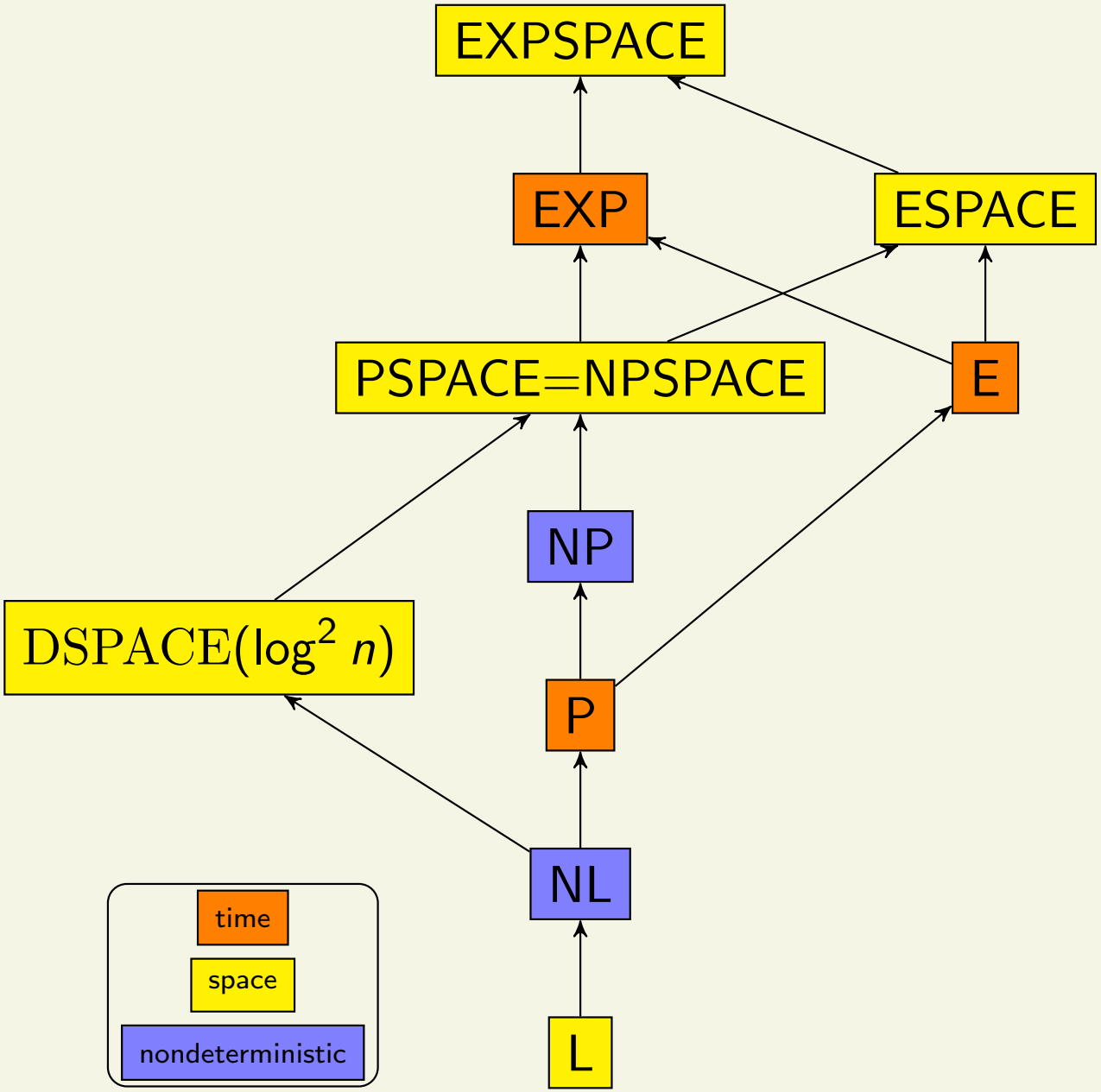
1. $\mathrm{P} = \mathrm{NP}$
2. *There is a polynomial-time algorithm A that for any formula ϕ outputs a satisfying assignment if ϕ is satisfiable, or "unsatisfiable" if it is not.*

Time and Space Classes Summary





Nondeterminism - NP and NL



Open problems:

- P = NP?
- P = PSPACE?
- NP = PSPACE?
- PSPACE = EXP?
- NP = EXP?
- NP ⊆ E? E ⊆ NP?
- PSPACE ⊆ E?
  E ⊆ PSPACE?
- L = NL?
- NL = P?
- NL = NP?
- L = NP?

Known:

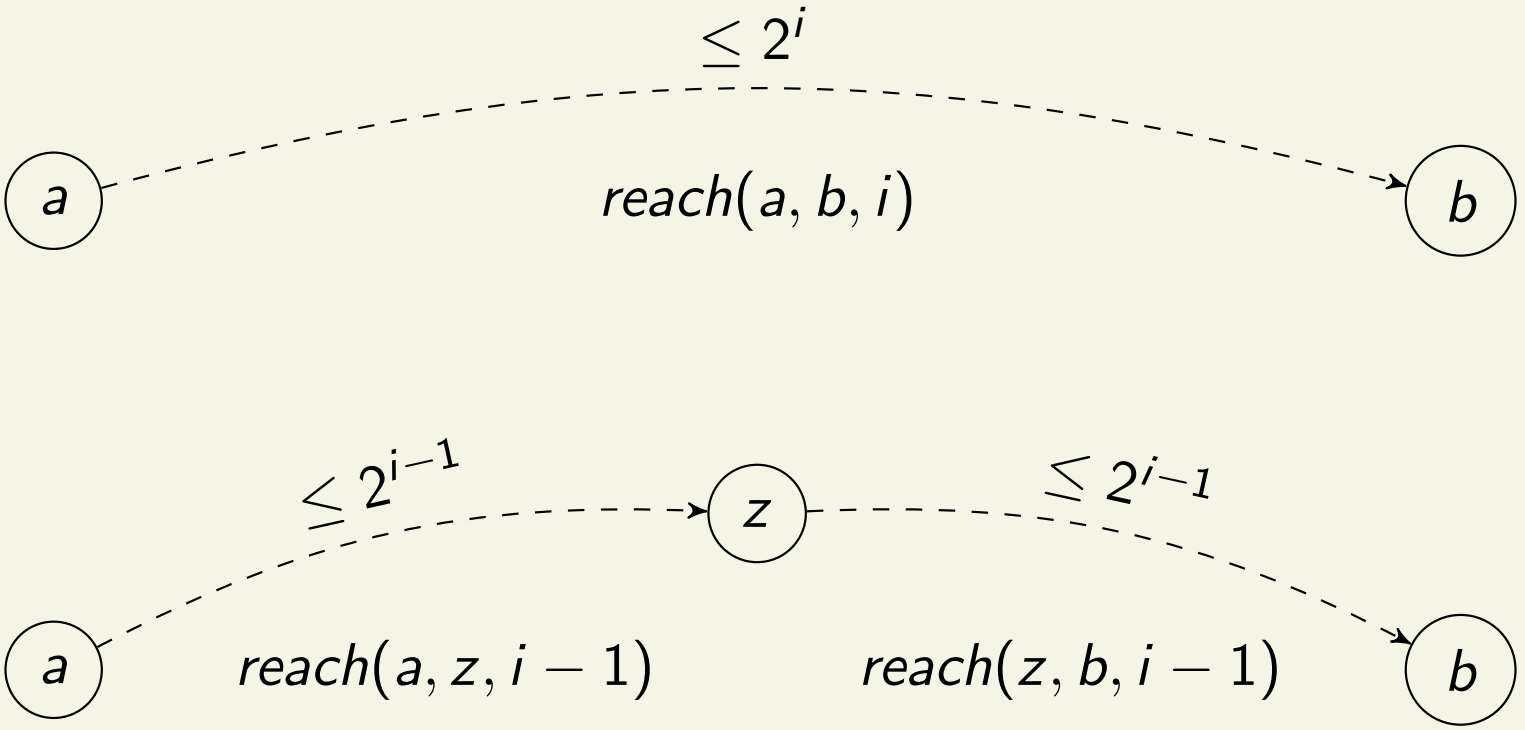
- P ≠ E ≠ EXP
- L ≠ DSPACE($\log^2$ n)
  ≠ PSPACE
  ≠ ESPACE
  ≠ EXPSPACE

Savitch's Algorithm

$reach(a, b, i)$ is true if there is a path $a \to b$ of length at most 2^i

Notice that

$$reach(a, b, i) \iff (\exists z)\ reach(a, z, i-1) \wedge reach(z, b, i-1).$$



Hierarchy Theorems

Time Hierachy Theorem (Hartmanis and Stearns, 1965)

Let $t_1, t_2 : \mathbb{N} \to \mathbb{N}$ be functions with

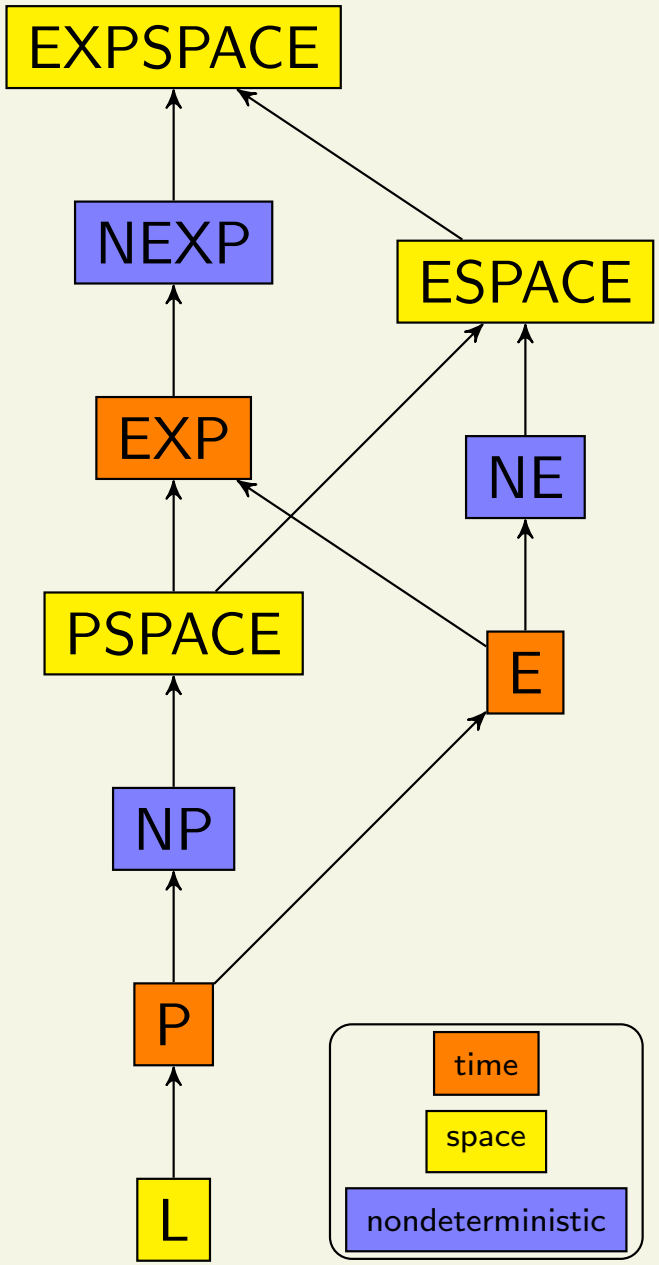
- $t_1(n) \log t_1(n) = o(t_2(n))$,
- t_2 is time-constructible,
- and $t_1(n) = \Omega(n)$.

Then $\mathrm{DTIME}(t_1(n)) \subsetneq \mathrm{DTIME}(t_2(n))$.

If t_2 is sufficiently larger than t_1, then there are problems that can be solved in t_2 time that cannot be solved in t_1 time.

Also:

- Space Hierarchy Theorem
- Nondeterministic Time Hierarchy Theorem



In this diagram:

- the time classes are different,
- the space classes are different, and
- the nondeterministic classes are different.

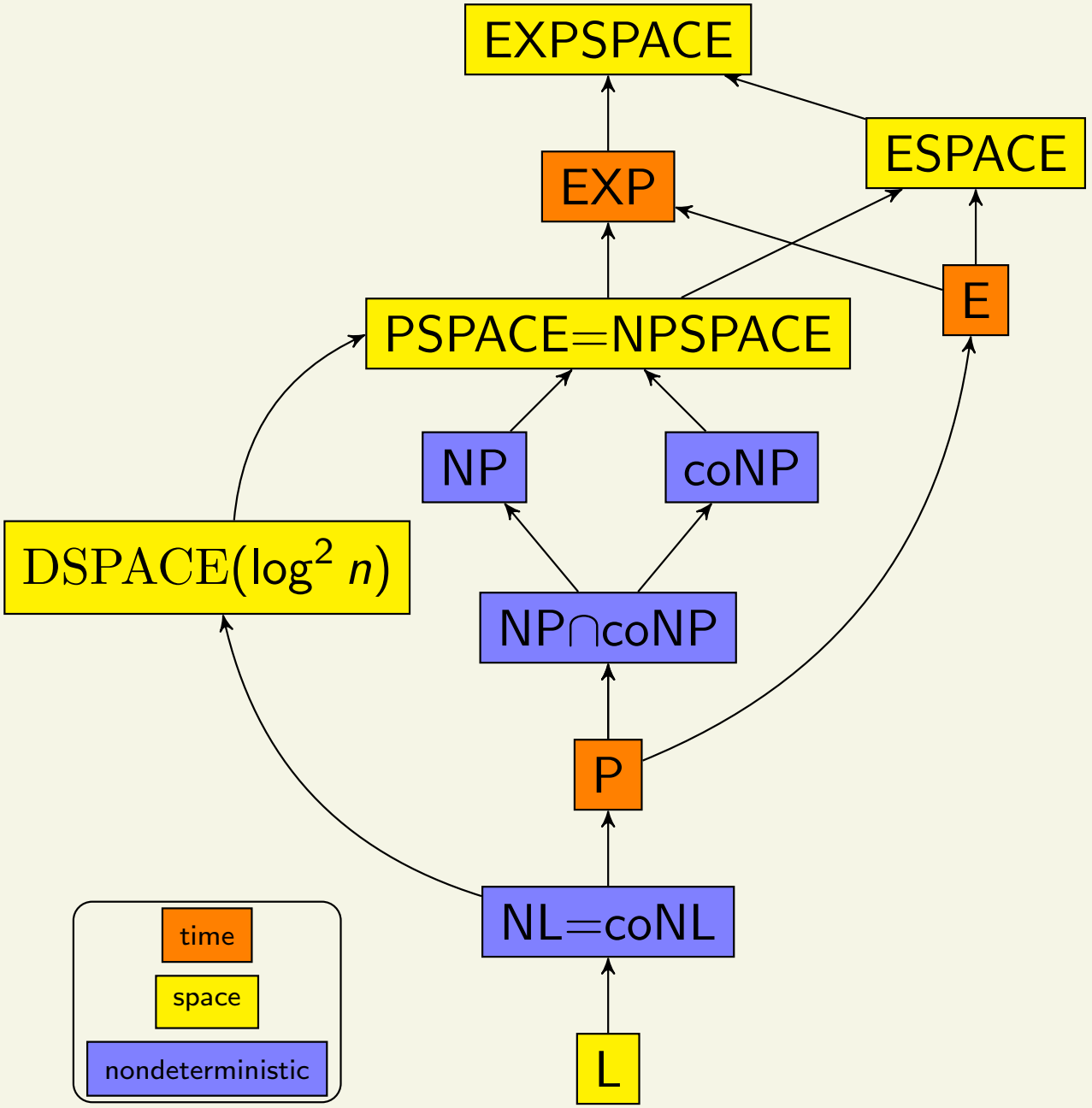
Most other relationships between these classes are open questions.

We know $\mathrm{NP} \neq \mathrm{NEXP}$. This tells us at least one of

- $\mathrm{NP} \neq \mathrm{PSPACE}$,
- $\mathrm{PSPACE} \neq \mathrm{EXP}$, or
- $\mathrm{EXP} \neq \mathrm{NEXP}$

must be true. However, it is an open problem to prove any of these statements!

NP vs. coNP and NL=coNL



Open problems:

- P = NP?
- P = PSPACE?
- NP = PSPACE?
- PSPACE = EXP?
- NP = EXP?
- NP ⊆ E? E ⊆ NP?
- PSPACE ⊆ E?
  E ⊆ PSPACE?
- L = NL?
- NL = P?
- NL = NP?
- L = NP?
- NP = coNP?
- P = NP ∩ coNP?

Known:

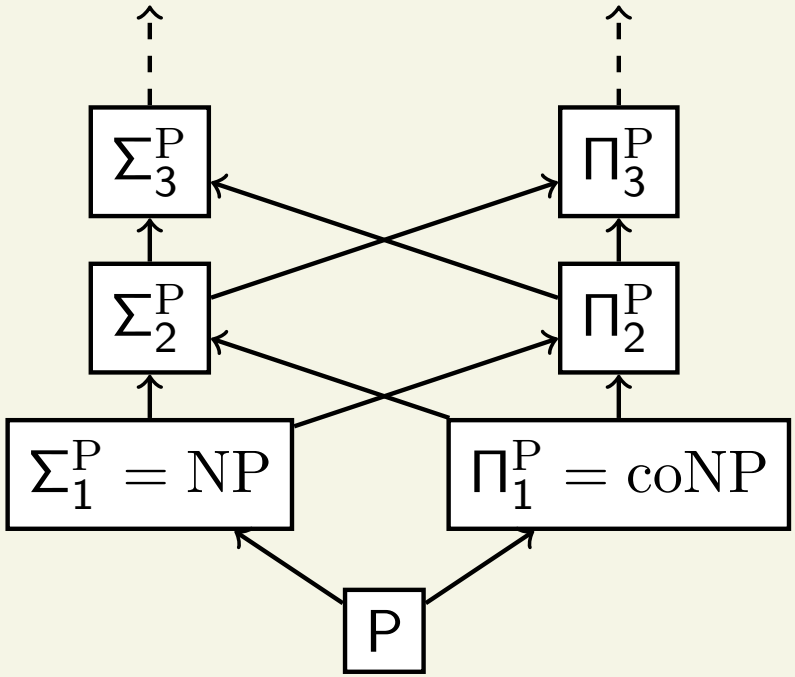
- P ≠ E ≠ EXP
- L ≠ DSPACE(log^2 n)
  ≠ PSPACE
  ≠ ESPACE
  ≠ EXPSPACE

PSPACE-Complete Problems

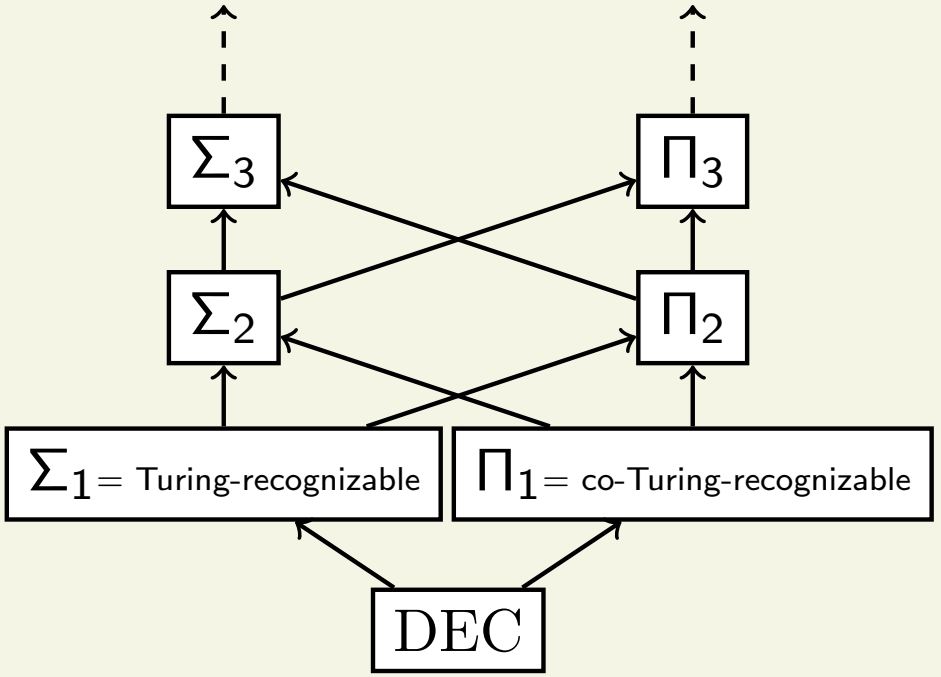
- TQBF
- Generalized Geography (GG)
- Many other generalized games
  - Tic-tac-toe
  - Othello
  - Chess
  - Go
- Problems about NFAs and regular expressions:
  - $ALL_{\text{NFA}} = \{\langle N \rangle \mid N \text{ is an NFA with } L(N) = \Sigma^*\}$
  - $EQ_{\text{NFA}} = \{\langle M, N \rangle \mid M, N \text{ are NFAs with } L(R) = L(S)\}$
  - $ALL_{\text{REGEX}} = \{\langle R \rangle \mid R \text{ is a regular expression with } L(R) = \Sigma^*\}$
  - $EQ_{\text{REGEX}} = \{\langle R, S \rangle \mid R, S \text{ are regular expressions with } L(R) = L(S)\}$

The Polynomial-Time Hierarchy

The Polynomial-Time Hierarchy is a collection of complexity classes that sit about NP and coNP. It is an analogue of the Arithmetical Hierarchy.



The Polynomial-Time Hierarchy



The Arithmetical Hierarchy

PSPACE = AP
TQBF
PH
Σ_4^P
TQBF$_4$
Π_4^P
Σ_3^P
TQBF$_3$
Π_3^P
Σ_2^P
TQBF$_2$
UNIQUE-SAT
Π_2^P
MIN-FORMULA
Σ_1^P = NP
SAT
Π_1^P = coNP
TAUT
$\Sigma_0^P = \Pi_0^P = P$
∃∀∃
∀∃∀
∃∀
∀∃
∃
∀

Approximation Algorithms

Let X be a minimization problem.
An algorithm $\mathcal{A}$ is an $\alpha(n)$-approximation algorithm if for all n, for all instances of size n, $\mathcal{A}$ returns a solution with value at most $\alpha(n)$ times the value of the optimal solution.

For example:

- a $\log n$-approximation algorithm
- a 2-approximation algorithm

Let X be a maximization problem.
An algorithm $\mathcal{A}$ is an $\alpha(n)$-approximation algorithm if for all n, for all instances of size n, $\mathcal{A}$ returns a solution with value at least $\alpha(n)$ times the value of the optimal solution.

For example:

- a $\frac{7}{8}$-approximation algorithm
- a $\frac{\log n}{n}$-approximation algorithm

Approximating MAX3SAT

Theorem

There is a polynomial-time $\frac{7}{8}$-approximation algorithm for MAX3SAT.

Theorem (Hästad, 1997)

If $\mathrm{P} \neq \mathrm{NP}$, then for every $\epsilon > 0$, there is no polynomial-time $(\frac{7}{8} + \epsilon)$-appoximation algorithm for MAX3SAT.

Approximating TSP

Theorem

If $\mathrm{P} \neq \mathrm{NP}$, *then for any polynomial-time computable function* $\alpha(n)$, *there is no* $\alpha(n)$*-approximation algorithm for TSP.*

Theorem (Christofides, 1976)

There is a polynomial-time $\frac{3}{2}$-approximation algorithm for Metric TSP.

Which problems can be solved on a computer? What is a computer?

Four types of algorithms – models of computation:

1. **Finite-State Automata**: algorithms with extremely limited memory
2. **Pushdown Automata**: algorithms that have a stack data structure
3. **Turing Machines**: any algorithm can be implemented on a Turing machine according to the *Church-Turing thesis*
4. **Polynomial-Time Algorithms**: efficient computation