If we can *verify a solution* efficiently,
can we also *find a solution* efficiently?

Does $P = NP$?

NP-complete Problems

play a key role in whether $P = NP$

- Hilbert/Cook
  - History
- P, NP, NPC
  - Definitions
- Reductions
  - Strategy

The Entscheidungsproblem (Decision Problem)

Hilbert lecture (1900) - 23 unsolved mathematics problems

Problem 10: Determine solvability of a Diophantine equation

Equations with integer coefficients and integer solutions

Best known example: $x^n + y^n = z^n$ (Fermat's last theorem)

Shakeups: Quantum mechanics | Russell's paradox

Hilbert sought the rigorous axiomatisation of mathematics

Entscheidungsproblem: can we find a decision procedure to determine the provability of any well-formed formula

Published in 1928 with Willhelm Ackermann

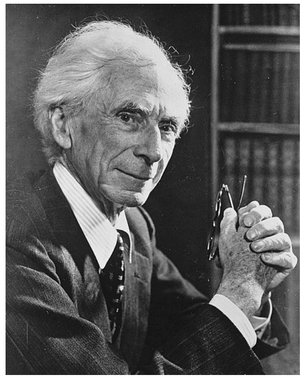


Image credits/References:
(Hilbert picture) https://en.wikipedia.org/wiki/David_Hilbert#/media/File:Hilbert.jpg
(Minkowski picture) https://en.wikipedia.org/wiki/Hermann_Minkowski#/media/File:De_Raum_zeit_Minkowski_Bild_(cropped).jpg
(Hilbert lecture) http://aleph0.clarku.edu/~djoyce/hilbert/problems.html
(Diophantus picture) https://en.wikipedia.org/wiki/Diophantus
(Russell picture) https://en.wikipedia.org/wiki/Bertrand_Russell#/media/File:Bertrand_Russell_1957.jpg
C. Petzold, "The annotated Turing: a guided tour through Alan Turing's historic paper on computability and the Turing machine" (2008)
(Ackermann) https://en.wikipedia.org/wiki/Wilhelm_Ackermann#/media/File:Ackermann_Wilhelm.jpg

Gödel And Herbrand

Gödel published "On Formally Undecidable Propositions of Principia Mathematica Systems I" (1931)

Introduced Gödel's Incompleteness theorem

Consistency proof for arithmetic within system is impossible

Hilbert's response was "somewhat angry"

A decision process for the truth of a formula was impossible

A decision process for provability was still possible

Princeton lectures on general recursive functions (1934)

Gödel credited the idea to Jacques Herbrand

With incompleteness theorem, implicitly ruled out a solution

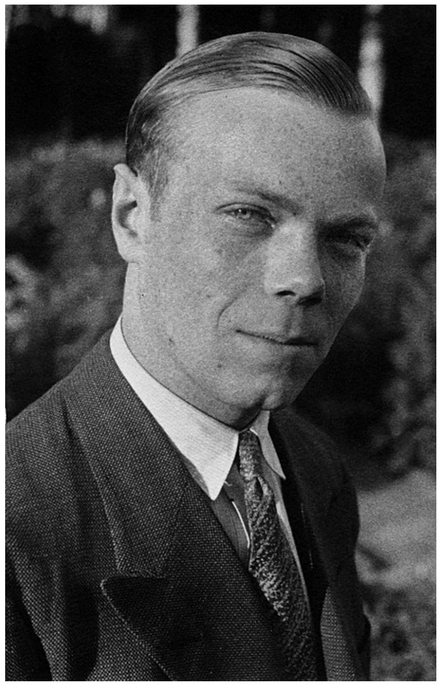
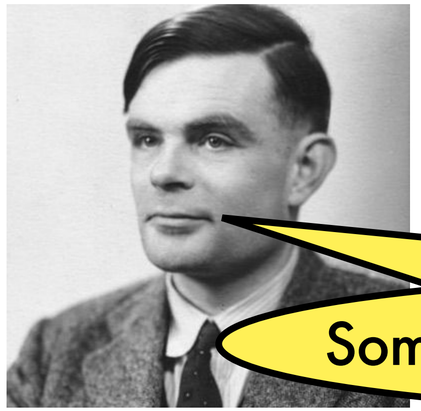


Image credits/References:
https://en.wikipedia.org/wiki/Kurt_Gödel#/media/File:Kurt_gödel.jpg
(Herbrand image) https://en.wikipedia.org/wiki/Jacques_Herbrand#/media/File:J_Herbrand_1931.jpg
(Background story) J. Erickson, http://jeffe.cs.illinois.edu/teaching/algorithms/models/06-turing-machines.pdf (2016)

Church And Turing



In 1936, Church & Turing showed no general procedure could be found to decide if an arbitrary proposition is provable from axioms of first order logic

Turing's paper contributed:

- *Turing machines* - a simple formal model of mechanical computation
- Proof that no Turing machine can solve the "halting problem" - decide whether a given Turing machine will halt or run indefinitely
- (Entscheidungsproblem) Proof that no Turing machine can decide provability of an arbitrary proposition

Some problems cannot be solved at all!

Suppose `halts(f)` returns `True` if `f` halts, `False` otherwise

```
def turing_mischief():
    if halts(turing_mischief):
        loop_forever()
```

contradiction!

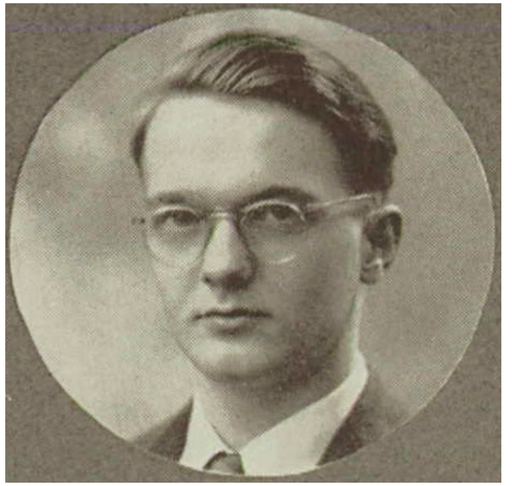
Does `turing_mischief()` halt?

`halts()` cannot exist!

Our focus: how efficiently can we solve a problem?

Image credits/References:
(Church picture) https://en.wikipedia.org/wiki/Alonzo_Church#/media/File:Alonzo_Church.jpg
(Turing picture) https://www.biography.com/scientists/alan-turing
A. Turing, "On computable numbers, with an application to the Entscheidungsproblem", J. of Mathematics (1936)
J. Erickson, http://jeffe.cs.illinois.edu/teaching/algorithms/models/06-turing-machines.pdf (2016)
C. Strachey, "An impossible program", The Computer Journal (1965)

Algorithmic Efficiency



Cobham's thesis

Efficient algorithms - solve problem in polynomial time

Runtime complexity $O(n^k)$ for some constant k and input size n

Decision problems: problems whose output is a boolean value (yes or no)

Class: P

Decision problems that can be solved in polynomial time

Class: NP

If answer is yes, proof can be checked in polynomial time

Class: co-NP

If answer is no, proof can be checked in polynomial time

Problems in P also in NP and co-NP

NP = "**N**ondeterministic **P**olynomial time" (equivalent definition)

Image credits/References:
A. Cobham, "The intrinsic computational difficulty of functions", (1965)
J. Erickson, Algorithms, http://jeffe.cs.illinois.edu/teaching/algorithms/ (2019)
(Cobham) https://recursed.blogspot.com/2014/11/alan-cobham-appreciation.html
https://en.wikipedia.org/wiki/NP_(complexity)

P Versus NP

Denied tenure ('70)

"It is to our everlasting shame that we were unable to persuade the math department to give him tenure."
Richard Karp

Basel turned down Euler...

Does P equal NP? (1971) ("P", "NP" due to Karp (1972))

The answer is no

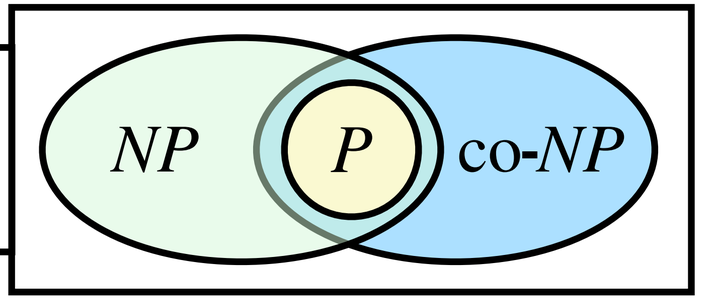
But no proof has been found that $P \neq NP$

It is the one of the 7 (1M USD) Millennium Prize problems

Limited progress in the form of "barrier" results

It is also unproven whether $NP \neq$ co-NP

What every reasonable human thinks the world looks like



"We can prove that we have no idea how to prove $P \neq NP$!"

Jeff Erickson

References/Image credits:
(Cook image) https://en.wikipedia.org/wiki/Stephen_Cook#/media/File:Stephen_A._Cook_1968_(enlarged_portion).jpg
S. Cook, "The complexity of theorem-proving procedures", ACM symposium on Theory of Computing (1971)
R. Karp, "Reducibility among combinatorial problems", Complexity of Computer Computations (1972)
J. Erickson, Algorithms, http://jeffe.cs.illinois.edu/teaching/algorithms/ (2019)
(sketch of Jeff Erickson by Damien Erickson) https://jeffe.cs.illinois.edu/
(R. Karp comment) https://www2.eecs.berkeley.edu/bears/CS_Anniversary/karp-talk.html

NP-complete And NP-hard

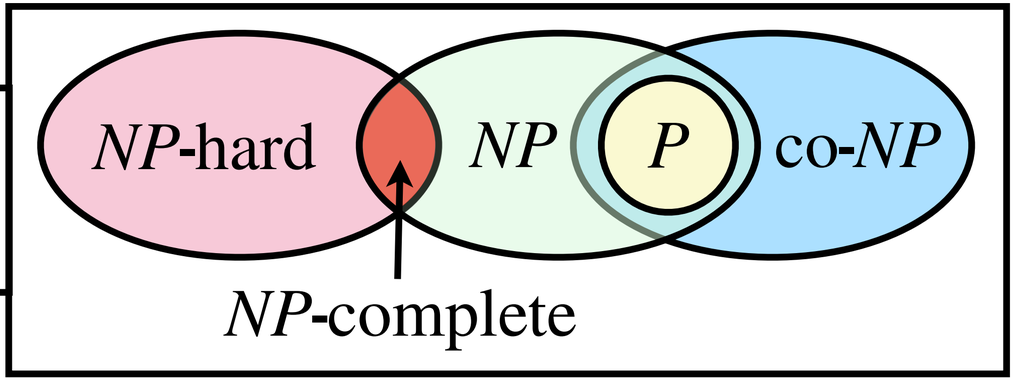
A problem is NP-hard if a polynomial-time algorithm for this problem implies a polynomial-time algorithm for all problems in NP

NP-hard problems are "at least as hard as every NP problem"

A problem is NP-complete if it is both in NP-hard and in NP

NP-complete problems are the "hardest problems in NP"

What every reasonable human thinks the world looks like



Why is it useful to know that a problem is NP-complete?

Give up searching for a fast, exact solution (focus on approximation algorithm)

References:
J. Erickson, Algorithms, http://jeffe.cs.illinois.edu/teaching/algorithms/ (2019)
T. Cormen et al., "Introduction to algorithms", Chap 34, MIT press (2022)

Strategy To Show NP-completeness

Strategy: Optimisation vs decision problems | Reductions | A first NP-complete problem

The NP-complete definition applies to decision problems with "yes" or "no" answers

Many problems that we care about are optimisation problems e.g. "find shortest path"

We can often convert an optimisation problem into a decision problem:

Given an undirected graph G, vertices s and t, and integer k, does there exist a path in G between s and t consisting of at most k edges? a decision problem

If the decision problem variant is difficult, we can often show optimisation problem is difficult

References:
T. Cormen et al., "Introduction to algorithms", Chap 34, MIT press (2022)

Polynomial-time Reductions

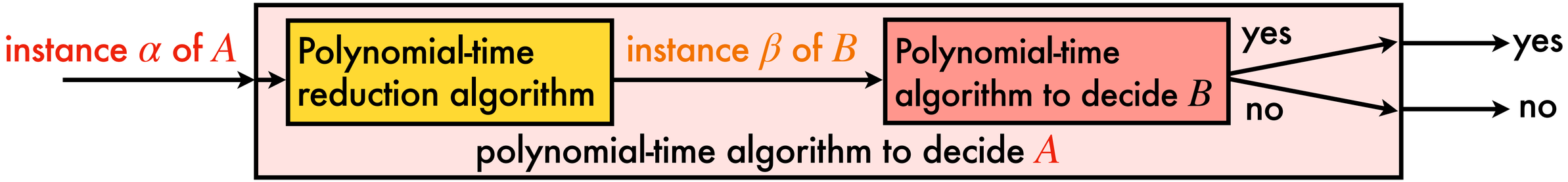
We'll refer to a specific configuration of a problem as an "instance" of that problem

Suppose we can transform instance α of problem A into instance β of problem B such that:

- transformation takes polynomial time
- answer for α is "yes" if and only if answer for β is "yes"

Polynomial-time reduction algorithm

This can allow us to decide A in polynomial time if we can decide B in polynomial time



Suppose we have: NP-complete problem A | Polynomial-time reduction algorithm $A \rightarrow B$

Then we know that B is NP-complete! | Proof by contradiction (could decide A in polynomial time)

References:
T. Cormen et al., "Introduction to algorithms", Chap 34, MIT press (2022)

A First NP-complete Problem

To use reduction to show a problem is NP-complete:

We need a "first" problem that we know is NP-complete

In 1971, Cook proved that the circuit-satisfiability problem is NP-complete

The same result was obtained independently by Leonid Levin (as a PhD student)

This result is known as the Cook-Levin theorem

Circuit-satisfiability problem:

Input: a boolean circuit of AND, OR and NOT gates

Question: does there exist a set of boolean inputs that causes the output to be 1?

References/image credits:
T. Cormen et al., "Introduction to algorithms", Chap 34, MIT press (2022)
J. Erickson, Algorithms, http://jeffe.cs.illinois.edu/teaching/algorithms/ (2019)
https://en.wikipedia.org/wiki/Cook–Levin_theorem
(Levin image) https://www.cs.bu.edu/~lnd/

Problems Solvable In Polynomial Time

Earlier, we described polynomial-time algorithms as "efficient" (Cobham's thesis)

Reasons that problems with polynomial-time solutions are considered tractable:

- Few problems with polynomial-time algorithms have very high order (e.g. $\Theta(n^{1000})$) rare in practice
- Once a first polynomial-time algorithm is found, more efficient variants are often found later
- Polynomial-time solvable problems in one computation model are often polynomial-time in others:
  - same problems are solvable in polynomial-time on Turing machines and serial RAM machines
- The class of polynomial-time solvable problems has several useful closure properties:
  - polynomials are closed under addition, multiplication and composition
  - can feed one polynomial-time algorithm into another to get algorithm that's still polynomial-time

References:
A. Cobham, "The intrinsic computational difficulty of functions", (1965)
T. Cormen et al., "Introduction to algorithms", Chap 34, MIT press (2022)

Formal Definitions: Problems And Encodings

Abstract decision problem: a mapping problem instances, I $\rightarrow$ $\{0,1\}$

Example: suppose $i = (G, s, t, k)$ is an instance of the abstract decision problem PATH

graph | nodes | number of edges

Then: $\text{PATH}(i) = 1$ if $\exists$ a path in G from s to t with $\leq k$ edges | $\text{PATH}(i) = 0$ otherwise

To solve an abstract decision problem instance, we need a way to communicate it to the computer

Encoding: a mapping abstract objects $\rightarrow$ binary strings

Example: we can encode natural numbers $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$ as strings $\{0, 1, 10, 11, \ldots\}$

The size of problem instance i is the length of its encoded string, denoted $|i|$

A problem whose instance set is the set of binary strings is a concrete problem

References:
D. Wilmer, Graduate Algorithms, Lecture 36, http://www.cs.cmu.edu/afs/cs/academic/class/15750-s17/ScribeNotes/lecture36.pdf (2017)
T. Cormen et al., "Introduction to algorithms", Chap 34, MIT press (2022)

A Definition For Complexity Class P

An algorithm solves a concrete problem in $T(n)$ time if:

Given problem instance i of size $n = |i|$, the algorithm produces a solution in $T(n)$ time

A concrete problem is polynomial-time solvable if there exists an algorithm to solve it in $O(n^k)$ time for some constant k

Complexity class $P = \{$concrete decision problems that are polynomial-time solvable$\}$

A subtle point: the choice of encoding affects the size of the problem instance

We can typically convert between "sensible" encodings in polynomial time (rules out unary encoding)

Consequently, encoding choice tends not to affect whether a problem is in P

$\langle o \rangle$ denotes "standard" encoding of o (e.g. code for integer polynomially related to binary repr. etc.)

References:
D. Wilmer, Graduate Algorithms, Lecture 36, http://www.cs.cmu.edu/afs/cs/academic/class/15750-s17/ScribeNotes/lecture36.pdf (2017)
T. Cormen et al., "Introduction to algorithms", Chap 34, MIT press (2022)

Formal-Language Theory Definitions

An alphabet Σ is a finite set of symbols

A language L over Σ is any set of strings formed of symbols from Σ

Example: $\Sigma = \{0, 1\}$, $L = \{10, 11, 101, 111, \ldots\}$ is the language of prime numbers in binary

ε: empty string

Σ^*: language of all strings over Σ

Example: $\Sigma = \{0, 1\}$, $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, \ldots\}$

Set-theoretic operations on languages (e.g. union and intersection) follow directly from definitions

The set of problem instances for decision problem Q is the language Σ^* where $\Sigma = \{0,1\}$

Q is fully characterised by the set of problem instances that produce a "yes" answer

We can interpret Q as a language L over $\Sigma = \{0,1\}$ where $L = \{x \in \Sigma^* : Q(x) = 1\}$

References:
D. Wilmer, Graduate Algorithms, Lecture 36, http://www.cs.cmu.edu/afs/cs/academic/class/15750-s17/ScribeNotes/lecture36.pdf (2017)
T. Cormen et al., "Introduction to algorithms", Chap 34, MIT press (2022)

Formal-Language Notation

The decision problem PATH has language:

$\text{PATH} = \{\langle G, s, t, k\rangle : G \text{ undirected graph}, \text{ nodes } s, t, \; k \in \mathbb{N}, \; \exists \text{path } s \rightsquigarrow t \text{ with at most k edges}\}$

encoded as binary strings

Formal languages succinctly express link between decision problems and algorithms that solve them

Algorithm A accepts a string $x \in \{0,1\}^*$ if, given x as input, the output is $A(x) = 1$

Algorithm A rejects a string $x \in \{0,1\}^*$ if, given x as input, the output is $A(x) = 0$

The language accepted by A is the set of accepted strings: $L = \{x \in \{0,1\}^* : A(x) = 1\}$

Note: even if A accepts L, we can't be sure that A rejects every $x \notin L$ (it may loop forever)

L is decided by A if every binary string in L is accepted and every binary string not in L is rejected

References:
D. Wilmer, Graduate Algorithms, Lecture 36, http://www.cs.cmu.edu/afs/cs/academic/class/15750-s17/ScribeNotes/lecture36.pdf (2017)
T. Cormen et al., "Introduction to algorithms", Chap 34, MIT press (2022)

Alternative Definition Of P

L is accepted in polynomial time by A if L is accepted by A and there exists a constant k s.t. for any string $x \in L$ with $|x| = n$, A accepts x in $O(n^k)$ time

L is decided in polynomial time by A if there exists a constant k s.t. for any string $x \in \{0,1\}^*$ with $|x| = n$, A decides x in $O(n^k)$ time

Key difference:

- To accept a language L, algorithm A only needs to provide an answer for strings in L
- To decide a language L, algorithm A must accept/reject every string in $\{0,1\}^*$

For Turing's Halting Problem, an accepting algorithm exists, but no decision algorithm exists

Complexity class: a set of languages whose membership is determined by a complexity measure (e.g. runtime) of an algorithm that decides the language

Complexity class $P = \{L \subseteq \{0,1\}^* :$ there exists an algorithm that decides L in polynomial time$\}$

References:
D. Wilmer, Graduate Algorithms, Lecture 36, http://www.cs.cmu.edu/afs/cs/academic/class/15750-s17/ScribeNotes/lecture36.pdf (2017)
T. Cormen et al., "Introduction to algorithms", Chap 34, MIT press (2022)

Hamiltonian And Eulerian Cycles

A hamiltonian cycle of undirected graph $G = (V, E)$ is a simple cycle containing every vertex $v \in V$

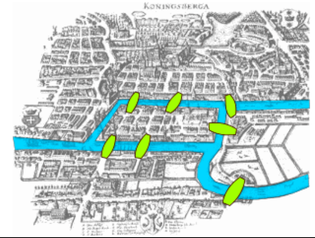
An eulerian cycle of undirected graph $G = (V, E)$ is a cycle containing every edge $e \in E$ once

The decision problems of determining whether G contains such cycles are defined by languages:

Hamilton's game

$$\text{HAM\_CYCLE} = \{\langle G \rangle : \text{G contains a hamiltonian cycle}\}$$

$$\text{EULER\_CYCLE} = \{\langle G \rangle : \text{G contains an eulerian cycle}\}$$



Euler's bridge problem

Euler's theorem: An undirected graph has an eulerian cycle $\iff$ every vertex has even degree

So $\text{EULER\_CYCLE} \in P$

No known polynomial-time algorithm decides HAM_CYCLE (so $\text{HAM\_CYCLE} \notin P$)

References:
D. Wilmer, Graduate Algorithms, Lec. 36, http://www.cs.cmu.edu/afs/cs/academic/class/15750-s17/ScribeNotes/lecture36.pdf (2017)
T. Cormen et al., "Introduction to algorithms", Chap 34, MIT press (2022)
(Hamilton's game) https://en.wikipedia.org/wiki/Icosian_game#/media/File:Hamiltonian_path_3d.svg
(Euler's bridge puzzle) https://en.wikipedia.org/wiki/Seven_Bridges_of_Königsberg

Polynomial-time Verification And NP

If someone *provided* us with a hamiltonian cycle, we could verify it in polynomial time:

Check the cycle is a valid permutation of the vertices in V and that each proposed edge exists in E

A verification algorithm A takes two arguments: input binary string x and certificate binary string y

Algorithm A verifies string x if there exists certificate y such that $A(x, y) = 1$

The language verified by A is $L = \{x \in \{0,1\}^* : \exists y \in \{0,1\}^* : A(x, y) = 1\}$

Complexity class NP is the class of languages that can be verified by a polynomial-time algorithm

$L \in NP \iff \exists$ polynomial-time A and constant c such that

$$L = \{x \in \{0,1\}^* : \exists \text{ certificate } y \text{ with } |y| = O(|x|^c) \text{ s.t. } A(x, y) = 1\}$$

$L \in$ co-$NP \iff \exists$ polynomial-time A and constant c such that

$$L = \{x \in \{0,1\}^* : \exists \text{ certificate } y \text{ with } |y| = O(|x|^c) \text{ s.t. } A(x, y) = 0\}$$

Little has been proven about the relationships between P, NP and co-NP

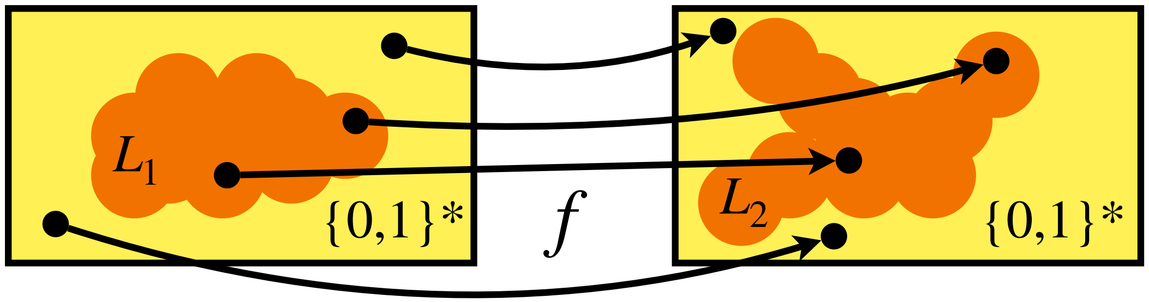
References:
D. Wilmer, Graduate Algorithms, Lec. 36, http://www.cs.cmu.edu/afs/cs/academic/class/15750-s17/ScribeNotes/lecture36.pdf (2017)
T. Cormen et al., "Introduction to algorithms", Chap 34, MIT press (2022)

Reducibility

One strategy to solve a problem is to reduce (or recast) it to another problem that we can solve

In the formal-language decision problem framework:

L_1 is polynomial-time reducible to L_2, denoted $L_1 \leq_P L_2$, if there exists a polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that $\forall x \in \{0,1\}^*,\ x \in L_1 \iff f(x) \in L_2$



f reduces L_1 to L_2

Lemma: If $L_1, L_2 \subseteq \{0,1\}^*$ are languages such that $L_1 \leq_P L_2$ then $L_2 \in P \implies L_1 \in P$

Proof: Construct a polynomial-time algorithm F to compute $f(x)$ for any input x

Let A_2 be a polynomial-time algorithm that decides L_2 $A_2 \circ F$ decides L_1 in polynomial time

References:
D. Wilmer, Graduate Algorithms, Lec. 36, http://www.cs.cmu.edu/afs/cs/academic/class/15750-s17/ScribeNotes/lecture36.pdf (2017)
T. Cormen et al., "Introduction to algorithms", Chap 34, MIT press (2022)

NP-completeness

A language $L \subseteq \{0,1\}^*$ is NP-hard if $L' \leq_P L$ for every $L' \in NP$

A language $L \subseteq \{0,1\}^*$ is NP-complete if:

1. $L \in NP$, and

2. $L' \leq_P L$ for every $L' \in NP$

NPC is the complexity class of NP-complete languages

Why is NP-completeness so critical to the study of whether $P = NP$?

Theorem: If any NP-complete problem is polynomial-time solvable, then $P = NP$

Proof: Suppose $L \in P$ in addition to $L \in NPC$.

By definition, we know that $L' \leq_P L$ for any $L' \in NP$, so $L' \in P$ $\Longrightarrow P = NP$

This is why NP-complete problems receive a lot of attention

References:
D. Wilmer, Graduate Algorithms, Lec. 36, http://www.cs.cmu.edu/afs/cs/academic/class/15750-s17/ScribeNotes/lecture36.pdf (2017)
T. Cormen et al., "Introduction to algorithms", Chap 34, MIT press (2022)