# 2 Complexity Theory Recap

*22 April 2025*

Prof. Dr. Sebastian Wild

CS627 (Summer 2025)
Philipps-Universität Marburg
*version* `2025-05-12 18:16 H`

# Outline

## 2 Complexity Theory Recap

## 2.1 P and NP Informally

# Hard problems

- Some algorithmic problems are **"hard nuts" to crack.**
  - e. g., the *Traveling Salesperson Problem (TSP):*

    Given: $n$ cities $S_1, \ldots, S_n$,
    all $n(n-1)$ pairwise
    distances $d(S_i, S_j) \in \mathbb{N}$ $\quad (i \neq j)$

    Goal: Shortest round trip through all cities

    always exact, always correct    polytime
  - no general, efficient algorithm known!
    (despite decades of intensive research . . . )

$\rightsquigarrow$ It *seems* as if there is no efficient algorithm for TSP!

- **But:** can we *prove* that? 
- Despite similarly intensive research: **No!** (not yet)

Doesn't sound like a shining example for theoretical computer science? . . . stay tuned!

# United in incapacity

*"I can't find an efficient algorithm, but neither can all these famous people."*

Garey, Johnson 1979

# Complexity Theory

- *Complexity theory* allows us to *compare* the *hardness* of algorithmic problems.

$$\leq_p$$

$A$: old problem
**Consensus: hard**

$B$: new problem
**Status: unknown**
(seems hard to *us* ...)

**Intuitive idea:**

1. If $A$ is a known hard nut, and
2. $\boxed{B \text{ is at least as hard as } A,}$

then $B$ is a hard nut, too!

**Formally:**

efficient = polytime

1. $A$ is NP-hard: probably $\nexists$ eff. alg. for $A$
2. $A \leq_p B$: $\exists$ eff. alg. for $B \implies \exists$ eff. alg. for $A$

$\leadsto$ $B$ is NP-hart: probably $\nexists$ eff. alg. for $B$!

# P and NP – Intuitive Synopsis

- $\boxed{\text{P = class of problems for which there is an algorithm } A \text{ and a polynomial } p \text{ such that } A \textbf{ solves} \text{ every instance } I \text{ in time } O(p(|I|)).}$
  - P for "polynomial" – i. e., all problems where a solution can be *found* by a (deterministic) algorithm in polynomial time.
- $\boxed{\text{NP = class of problems for which there is an algorithmus } A \text{ and a polynomial } p \text{ such that } A \text{ can } \textbf{verify} \text{ a given candidate solution } l(I) \text{ of a given instance } I \text{ in time } O(p(|I|)), \text{ i. e., check whether } l(I) \text{ solves } I \text{ or not.}}$
  - NP for "nondeterministically polynomial" – i. e., all problems where a solution can be *found* by a *nondeterministic* algorithm in polynomial time.
  - This is equivalent to the above characterization via verification.
- We know $\mathsf{P} \subseteq \mathsf{NP}$. We *think* $\mathsf{P} \subsetneq \mathsf{NP}$, i. e., $\mathsf{P} \neq \mathsf{NP}$.

  The question "$\mathsf{P} = \mathsf{NP}$?" is one of the famous **millenium problems** and arguably the **most important open problem of theoretical computer science**.

## 2.2 Models of Computation

# Mathematical Models of Computation

- complexity classes talk about sets of problems based upon whether they allow an algorithm of a certain cost
- in general, this depends on the allowable algorithms and their costs!

$\rightsquigarrow$ need to fix a machine model

A *machine model* decides

- what algorithms are possible
- how they are described (= programming language)
- what an execution *costs*

**Goal:** Machine models should be
detailed and powerful enough to reflect actual machines,
abstract enough to unify architectures,
simple enough to analyze.

# Random Access Machines

Standard model for detailed complexity analysis:

## Random access machine (RAM)

more detail in §2.2 of *Sequential and Parallel Algorithms and Data Structures* by Sanders, Mehlhorn, Dietzfelbinger, Dementiev

- unlimited *memory* $\mathtt{MEM}[0], \mathtt{MEM}[1], \mathtt{MEM}[2], \ldots$
- fixed number of *registers* $R_1, \ldots, R_r$ (say $r = 100$)
- memory cells $\mathtt{MEM}[i]$ and registers $R_i$ store $w$-bit integers, i. e., numbers in $[0..2^w - 1]$
  $w$ is the word width/size; typically $\boxed{w \propto \lg n}$ $\rightsquigarrow$ $2^w \approx n$
- Instructions:
  - load & store: $R_i := \mathtt{MEM}[R_j]$ $\mathtt{MEM}[R_j] := R_i$
  - operations on registers: $R_k := R_i + R_j$ (arithmetic is *modulo* $2^w$!)
    also $R_i - R_j$, $R_i \cdot R_j$, $R_i \operatorname{div} R_j$, $R_i \bmod R_j$
    C-style operations (bitwise and/or/xor, left/right shift)
  - conditional and unconditional jumps
- time cost: number of executed instructions
- space cost: total number of touched memory cells

# RAM-Program Example

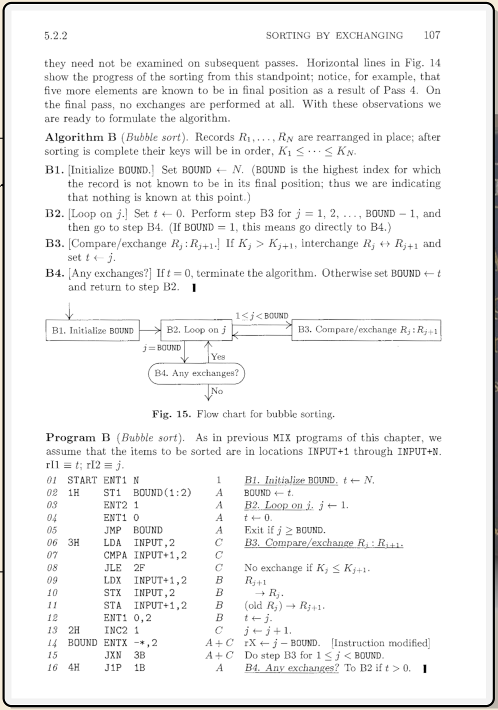
## Example RAM program

1 *// Assume:* $R_1$ *stores number* $N$
2 *// Assume:* $\texttt{MEM}[0..N)$ *contains list of* $N$ *numbers*
3 $R_2 := R_1$;
4 $R_3 := R_1 - 2$;
5 $R_4 := \texttt{MEM}[R_3]$;
6 $R_5 := R_3 + 1$;
7 $R_6 := \texttt{MEM}[R_5]$;
8 **if** $(R_4 \le R_6)$ goto line 11;
9 $\texttt{MEM}[R_3] := R_6$;
10 $\texttt{MEM}[R_5] := R_4$;
11 $R_3 := R_3 - 1$;
12 **if** $(R_3 \ge 0)$ goto line 5;
13 $R_2 := R_2 - 1$;
14 **if** $(R_2 > 0)$ goto line 4;
15 *// Done:*

# RAM-Program Example

## Example RAM program

1 // *Assume:* $R_1$ *stores number* $N$
2 // *Assume:* `MEM`$[0..N)$ *contains list of* $N$ *number*
3 $R_2 := R_1;$
4 $R_3 := R_1 - 2;$
5 $R_4 :=$ `MEM`$[R_3];$
6 $R_5 := R_3 + 1;$
7 $R_6 :=$ `MEM`$[R_5];$
8 **if** $(R_4 \le R_6)$ goto line 11;
9 `MEM`$[R_3] := R_6;$
10 `MEM`$[R_5] := R_4;$
11 $R_3 := R_3 - 1;$
12 **if** $(R_3 \ge 0)$ goto line 5;
13 $R_2 := R_2 - 1;$
14 **if** $(R_2 > 0)$ goto line 4;
15 // *Done:* `MEM`$[0..N)$ *sorted*





they need not be examined on subsequent passes. Horizontal lines in Fig. 14 show the progress of the sorting from this standpoint; notice, for example, that five more elements are known to be in final position as a result of Pass 4. On the final pass, no exchanges are performed at all. With these observations we are ready to formulate the algorithm.

**Algorithm B** (*Bubble sort*). Records $R_1, \ldots, R_N$ are rearranged in place; after sorting is complete their keys will be in order, $K_1 \le \cdots \le K_N$.

**B1.** [Initialize BOUND.] Set BOUND $\leftarrow N$. (BOUND is the highest index for which the record is not known to be in its final position; thus we are indicating that nothing is known at this point.)

**B2.** [Loop on $j$.] Set $t \leftarrow 0$. Perform step B3 for $j = 1, 2, \ldots,$ BOUND $- 1$, and then go to step B4. (If BOUND $= 1$, this means go directly to B4.)

**B3.** [Compare/exchange $R_j : R_{j+1}$.] If $K_j > K_{j+1}$, interchange $R_j \leftrightarrow R_{j+1}$ and set $t \leftarrow j$.

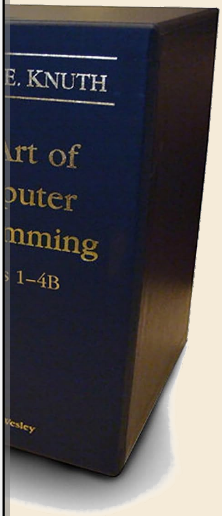
**B4.** [Any exchanges?] If $t = 0$, terminate the algorithm. Otherwise set BOUND $\leftarrow t$ and return to step B2. ▮



**Fig. 15.** Flow chart for bubble sorting.

**Program B** (*Bubble sort*). As in previous MIX programs of this chapter, we assume that the items to be sorted are in locations INPUT+1 through INPUT+N. rI1 $\equiv t$; rI2 $\equiv j$.

```
START  ENT1 N            1      B1. Initialize BOUND. t ← N.
1H     ST1  BOUND(1:2)   A      BOUND ← t.
       ENT2 1            A      B2. Loop on j. j ← 1.
       ENT1 0            A      t ← 0.
       JMP  BOUND        A      Exit if j ≥ BOUND.
3H     LDA  INPUT,2      C      B3. Compare/exchange R_j : R_{j+1}.
       CMPA INPUT+1,2    C
       JLE  2F           C      No exchange if K_j ≤ K_{j+1}.
       LDX  INPUT+1,2    B      R_{j+1}
       STX  INPUT,2      B        → R_j.
       STA  INPUT+1,2    B      (old R_j) → R_{j+1}.
       ENT1 0,2          B      t ← j.
2H     INC2 1            C      j ← j + 1.
BOUND  ENTX -*,2         A+C    rX ← j − BOUND.   [Instruction modified]
       JXN  3B           A+C    Do step B3 for 1 ≤ j < BOUND.
4H     J1P  1B           A      B4. Any exchanges? To B2 if t > 0. ▮
```



## 2.3 Turing Machines

# Keep it Simple, Stupid

- word-RAM (rather) realistic, but complicated
  - note that the machine has to grow with the inputs(!)
- for a coarse distinction of running time complexity, simpler models suffice
  - useful to reason about "all algorithms"
  - machine is fixed for all inputs sizes apart from storage for input

Many models of computation . . .

- $\mu$-recursive function
- Turing machines (TM)
- counter machines
- $\lambda$-calculus
- While-programs
- any Turing-complete language
- quantum computers
- . . .

. . . with strong equivalences:

1. all proven to lead to the *same* set of computable functions
2. **Church-Turing thesis**:
   any formalization of "effectively computable" is equivalent in this sense
3. ***Extended Church-Turing Thesis:***
   . . . and can be simulated with *polynomial overhead* on a TM
   - true for all on left . . .
   - except theoretical quantum computers!
   - ignore them for now

wake me when they exist

# Turing Machines

- invented by ***Alan Turing*** in 1936 as formalization for "computable by hand"

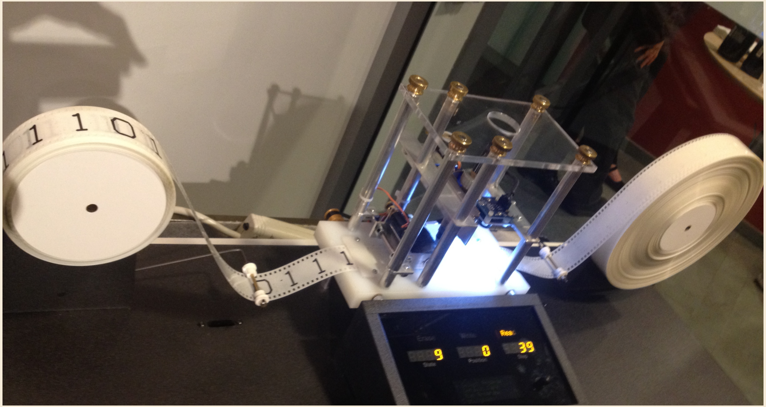
  In same paper, Turing proved undecidability of halting problem!

ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

By A. M. TURING.

[Received 28 May, 1936.—Read 12 November, 1936.]

- *minimalistic* model of universal computer, but can be built:

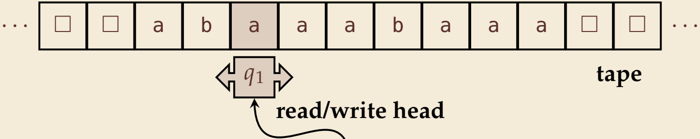


# Turingmaschinen in Circulation

# Turing Machines – Informal Recap

A *Turing machine* has

- a **finite control** via states
- an **input/output-tape**
  - unbounded length
  - initially contains input
  - all other cells contain "□"
- a **read/write head**
  - reads the current symbol
  - overwrites it with a new symbol
  - initially placed on beginning of input




| State | Tape Symbol | → | New Tape Symbol | Head Movement | New State |
|---|---|---|---|---|---|
| $q_0$ | a | → | a | right | $q_0$ |
| $q_0$ | b | → | b | right | $q_1$ |
| $q_0$ | □ | → | □ | none | $q_e$ |
| $q_1$ | a | → | b | right | $q_1$ |
| $q_1$ | b | → | b | right | $q_0$ |
| $q_1$ | □ | → | □ | none | $q_1$ |
| $q_e$ | – | → | — terminate the computation — | | |

**finite control**

# Turing Machines – Formal Syntax

## Definition 2.1 (Turing Machine (TM))

A *Turing machine* is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, \Box, q_{\text{halt}})$ with

- a finite set of *states* $Q$,
- an *input alphabet* $\Sigma$,
- a *tape alphabet* $\Gamma \supset \Sigma$,
- for deterministic TMs a *transition function* $\delta : (Q \setminus \{q_{\text{halt}}\}) \times \Gamma \to Q \times \Gamma \times \{L, R, N\}$
  for **non**deterministic TMs a *transition relation* $\delta : (Q \setminus \{q_{\text{halt}}\}) \times \Gamma \to 2^{Q \times \Gamma \times \{L,R,N\}}$
- an *initial state* $q_0 \in Q$,
- a *blank symbol* $\Box \in \Gamma \setminus \Sigma$, and
- a halting state $q_{\text{halt}} \in Q$

# Turing Machine – Computation Step

- Each step of a computation of TM $M$ has the form

  $\delta(q, a) = (q', b, d)$ resp. $\delta(q, a) \ni (q', b, d)$,

  with the semantics that

  - $M$ is in state $q \neq q_{\text{halt}}$
  - the cell below the read/write head currently contains symbol $a$
  - $M$ now changes (based on its finite control)
    - into state $q'$,
    - writes $b$ into the cell under the read/write head
    - and finally moves the read/write head in direction $d \in \{L, R, N\}$.
      ($L$ = left, $R$ = right, $N$ = none (stay))

- for deterministic TM $M$, $q$ and $a$ uniquely determine this action;
  for nondeterministic TM, we may have several possible actions.

- to formally define an entire computation, we have to encode the tape contents as well

# Turing Machines – Configurations

## Definition 2.2 (TM Configuration)

A *configuration (config)* of a TM $M$ is a string $C \in \Gamma^\star Q\, \Gamma^\star$.

The semantics of a config $C = \alpha q \gamma$, $q \in Q$, is tape content $\alpha\beta$ and head at first symbol of $\beta$.

## Definition 2.3 (TM Computation Relation)

The *computation relation* $\vdash$ is defined on the set of configurations of a TM $M$ as follows.

$$a_1 \dots a_m\, q\, b_1 \dots b_n \;\vdash\; \begin{cases} a_1 \dots a_m \quad q'\, c\, b_2 \dots b_n, & \delta(q, b_1) = (q', c, N),\ m \geq 0,\ n \geq 1, \\ a_1 \dots a_m\, c\ \, q'\, b_2 \dots b_n, & \delta(q, b_1) = (q', c, R),\ m \geq 0,\ n \geq 2, \\ a_1 \dots a_{m-1}\, q'\, a_m\, c\, b_2 \dots b_n, & \delta(q, b_1) = (q', c, L),\ m \geq 1,\ n \geq 1. \end{cases}$$

For the boundary case $n = 1$ and direction right, we set

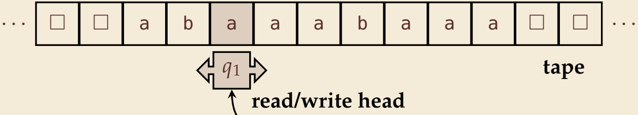
$$a_1 \dots a_m\, q\, b_1 \;\vdash\; a_1 \dots a_m\, c\, q'\, \Box \qquad \text{if } \delta(q, b_1) = (q', c, R),$$

For $m = 0$ and direction left, we similarly have set

$$q\, b_1 \dots b_n \;\vdash\; q'\, \Box c\, b_2 \dots b_n \qquad \text{if } \delta(q, b_1) = (q', c, L).$$

# Turing Machines – Configuration Example

**Example:**
For the shown TM $M$,
the current configuration is:
$C = \mathtt{ab}\, q_1\, \mathtt{aaabaaa}$




| State | Tape Symbol | $\longrightarrow$ | New Tape Symbol | Head Movement | New State |
|---|---|---|---|---|---|
| $q_0$ | a | $\longrightarrow$ | a | right | $q_0$ |
| $q_0$ | b | $\longrightarrow$ | b | right | $q_1$ |
| $q_0$ | □ | $\longrightarrow$ | □ | none | $q_e$ |
| $q_1$ | a | $\longrightarrow$ | b | right | $q_1$ |
| $q_1$ | b | $\longrightarrow$ | b | right | $q_0$ |
| $q_1$ | □ | $\longrightarrow$ | □ | none | $q_1$ |
| $q_e$ | – | $\longrightarrow$ | — terminate the computation — | | |

- TM Config $C = \alpha q \beta$ completely describes current state of computation
  - $\alpha\beta$ is the (non-blank) tape content
  - $q$ is the current state of the TM
  - the read/write head is on the first symbol of $\beta$

# Turing Machines – Computed Function

- With this setup, we can now formally define what a Turing machine computes.

## Definition 2.4 (Function computed by a TM)

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, \Box, q_{\text{halt}})$ by a TM. The function computed by $M$ on input $x \in \Sigma^\star$, written $M(x)$, is defined as
$M(x) = \{y : q_0 x \vdash^\star q_{\text{halt}} y\}$.
For deterministic TMs, we will also have $|M(x)| \leq 1$ and we write $M(x) = y$ for $M(x) = \{y\}$. ◀

## Definition 2.5 (Time and Space cost)

For a TM $M$ and input $x \in \Sigma^*$, we define
$time_M(x) = \inf\{t : q_0 x \vdash^t q_{\text{halt}} y\} \cup \{0\}$.
We define $space_M(x) = \inf\Big\{|\alpha\beta| : q_0 x \vdash^\star \alpha q \beta \vdash^\star q_{\text{halt}} y, |\alpha\beta|_\Box \leq 2\Big\} \cup \{0\}$. ◀

\# $\Box$ in $\alpha\beta$

- Note: *time* and *space* can be $\infty$ or $0$ for nondeterministic TMs.

# Turing Machines – Accepted Language

- Often convenient to use language acceptance instead of function computation.
- for deterministic TM, compute *characteristic function* of $L$: $\mathbb{1}_L(x) = \begin{cases} 1 & \text{if } x \in L \\ 0 & \text{otherwise} \end{cases}$
- care needed for nondeterministic TM

## Definition 2.6 (Language of TM)

The language $\mathcal{L}(M)$ accepted by a TM $M$ is defined as

$$\mathcal{L}(M) \quad = \quad \{w \in \Sigma^\star : 1 \in M(w)\}.$$

◀

$\rightsquigarrow$ $\boxed{\text{nondeterministic TM accepts } w \text{ iff some computation accepts } w}$

# Turing Machines – Several tapes

## Remark 2.7 ($k$-tape TMs)

We only consider one-tap TMs here. In general, $k$-tape TMs can be faster.
However, any language accepted by a $k$-tape TM in time $f(n)$
is also be accepted by a 1-tape TM with running time $O(f^2(n))$.
The models are thus *polynomially equivalent*. ◀

# Turing Machines – Totality

- In complexity theory, we will restrict ourselves to TMs that always halt.

### Definition 2.8 (Terminating TM)

A TM $M$ is *always terminating / total* if there is a function $T : \mathbb{N} \to \mathbb{N}$ such that $q_0 x = C_0 \vdash C_1 \vdash \cdots \vdash C_t$ implies $t \le T(|x|)$, and there is a $y$ such that $q_0 x \vdash^\star q_{\text{halt}} y$. ◀

- Note that in a terminating TM, we always have $1 \le time_M(x) \le T(|x|)$.

### Lemma 2.9 (Time-constrained TM)

Given a (potentially nonterminiating) TM $M$ and a function $T : \mathbb{N} \to \mathbb{N}$ **computable** in time $T(n)$, we can construct an *always terminating* TM $M'$ that simulates $M$ for $T(|x|)$ on $x$ and outputs `TIMEOUT` if $M$ has not terminated yet. Moreover, $time_{M'}(x) = O(T^2(|x|))$ for all $x \in \Sigma^\star$. ◀

$\rightsquigarrow$ If we are only interested in the (non)existence of a polynomial-time TM, we can restriction ourselves to total TMs that will never `TIMEOUT`.

# Models of Computation – Summary

- Concrete model such as TMs useful for some proofs
- Often, details do not matter as long as models are polynomially equivalent
  - Note: TM always means we are in the logarithmic cost model for arithmetic operations
- In the following, discuss more abstract notion of "algorithm"
  (fine to substitute by TM in each case)

## 2.4 The Classes P und NP

# Worst Case Complexity

## Definition 2.10 (Time and Space Complexity – Generic)

Let $\Sigma_I$ and $\Sigma_O$ two alphabets and $A$ an algorithm implementing a **total** mapping $\Sigma_I^\star \to \Sigma_O^\star$. Then for each $x \in \Sigma_I^\star$ we denote by $time_A(x)$ (resp. $space_A(x)$) the logarithmic time complexity (resp. logarithmic space complexity) for $A$ on $x$. ◀

Where needed, we can unpack this in full detail for Turing machines!

## Definition 2.11 (Worst-Case Complexity)

Let $\Sigma_I$ and $\Sigma_O$ be two alphabets and $A$ an algorithm implementing a **total** mapping $\Sigma_I^\star \to \Sigma_O^\star$. The *worst case time complexity of A* is the function $Time_A : \mathbb{N} \to \mathbb{N}$ with

$$Time_A(n) = \max\{time_A(x) : x \in \Sigma_I^n\},$$

for each $n \in \mathbb{N}$. The *worst case space complexity of A* is given by function $Space_A : \mathbb{N} \to \mathbb{N}$ with

$$Space_A(n) = \max\{space_A(x) : x \in \Sigma_I^n\}.$$

◀

# Decision Problems = Languages

## Definition 2.12 (Decision Problem and Algorithms)

A *decision problem* is given by $P = (L, U, \Sigma)$ for $\Sigma$ an alphabet and $L \subseteq U \subseteq \Sigma^\star$. An algorithm *$A$ solves (decides)* decision problem $P$, if for all $x \in U$

1. $A(x) = 1$ for $x \in L$, and
2. $A(x) = 0$ for $x \in U \setminus L$ (i. e., $x \notin L$)

holds. Here $A(x)$ denotes the output of $A$ on input $x$.
If $U = \Sigma^\star$ holds we denote $P$ briefly by $(L, \Sigma)$. ◀

$\rightsquigarrow$ $A$ computes a total function $A : U \to \{0, 1\}$,
the *characteristic function* $\mathbb{1}_L : U \to \{0, 1\}$ of language $L$.
We then write $L = \mathcal{L}(A)$, the language accepted by $A$.

We restrict our attention to ***decision problems***:
Given: $w \in \Sigma^\star$.
Goal: Is $w \in L$?

**Example:**
$w$ is an encoding of an instance of the traveling salesperson problem **and** a threshold $D$

$L = \{w : w \text{ encodes instance w/ opt. round trip length} \leq D\}$

# Optimal Algorithms

## Definition 2.13 (Upper/Lower Bounds, Optimal Algorithms)

Let $U$ be an algorithmic problem and $f, g$ functions $\mathbb{N}_0 \to \mathbb{R}^+$.

- We call $O(g(n))$ an *upper bound for time complexity of $U$* if there is an algorithm $A$ that solves $U$ in time $Time_A(n) \in O(g(n))$.
- We say $\Omega(f(n))$ is a *lower bound for time complexity of $U$* if every algorithm $A$ that solves $U$ needs time $Time_A(n) \in \Omega(f(n))$.
- An algorithm $A$ is called *optimal for $U$* if $Time_A(n) \in O(g(n))$ and $\Omega(g(n))$ is a lower bound for the time complexity of $U$.

# Running Time

## Definition 2.14 (time classes)

For function $f : \mathbb{N} \to \mathbb{N}$, the class $TIME(f(n))$ is the set of all languages $A$, for which there is a *deterministic* Turing machine $M$ with $\mathcal{L}(M) = A$ and $time_M(w) \le f(|w|)$ für alle $w \in \Sigma^\star$. ◀

## Definition 2.15 (P, tractable)

We define the class of languages $\mathsf{P}$ decidable in polynomial time by

$$\mathsf{P} \;:=\; \bigcup_{p \text{ polynomial}} TIME(p(n)).$$

A language (a decision problem) $L \in \mathsf{P}$ is called *tractable / efficiently decidable*. ◀

# Nondeterministic Running Time

Recall:

- A nondeterministic Turing machine / algorithm $M$ accepts $L$ ($\mathcal{L}(M) = L$) if for all $x \in L$ there is at least one computation of $M$ which accepts $x$ and for all $y \notin L$ every computation of $M$ rejects $y$.
- We only consider always-terminating Turing machines.
- The running time $time_M(x)$ of $M$ on $x$ is given by the longest computation of $M$ on $x$.

## Definition 2.16 (NTIME, NP)

For function $f : \mathbb{N} \to \mathbb{N}$, the class $NTIME(f(n))$ is the set of all languages $A$, for which there is a ***nondeterministic*** Turing machine $M$ with $\mathcal{L}(M) = A$ and $time_M(w) \le f(|w|)$ für alle $w \in \Sigma^\star$.
The class of languages $\mathsf{NP}$ is defined by

$$\mathsf{NP} := \bigcup_{p \text{ polynomial}} NTIME(p(n)).$$

◀

## 2.5 Nondeterminism = Verification

# Nondeterminism?

- The original definition of NP via nondeterministic Turing machines is not very intuitive.
- There is an equivalent characterization that is usually more convenient to use: Certificates and verifiers.

# Polynomially verifiable

### Definition 2.17 (Certificates, Verifier, VP)

Let $L \subseteq \Sigma^\star$ be a language.

- An algorithm $A$ acting on inputs from $\Sigma^\star \times \{0,1\}^\star$ is called *verifier for L* (notation $L = \mathcal{V}(A)$), if

$$L \quad = \quad \{w \in \Sigma^\star : \exists c \in \{0,1\}^\star \; A(w,c) = 1\}.$$

  If $A$ accepts input $(w, c)$ we say $c$ is *proof* or *certificate* for $w \in L$.

- A verifier $A$ for $L$ is a *polynomial-time verifier* if there is a $d \in \mathbb{N}$ such that for all $w \in L$, there is a proof $c$ (for $w \in L$) with $time_A(w,c) \in O(|w|^d)$.

- We define the class of polynomially verifiable languages $\mathsf{VP}$ by

$$\mathsf{VP} \quad = \quad \{\mathcal{V}(A) : A \text{ is polynomial time verifier}\}.$$

◀

# Nondeterminism ↔ certificate

**Theorem 2.18**

NP = VP.

## 2.6 Karp-Reductions und NP-Completeness

# Recap

- We restrict our attention to ***decision problems***:
  Given: $w \in \Sigma^\star$.
  Goal: Is $w \in L$?

**Example:**
$w$ is an encoding of an instance of the traveling salesperson problem **and** a threshold $D$

$L = \{w : w \text{ encodes instance w/ opt. round trip length} \leq D\}$

⇝ problems = (formal) languages $L \subseteq \Sigma^\star$

- problem instance = word $w \in \Sigma^\star$
- $w \in \Sigma^\star$ is a *Yes instance* if $w \in L$, otherwise a *No instance*

⇝ For problems on structures, e. g., graphs, we need an *encoding* of the instance as a string.
(often simple; standard data structures do the trick)

⇝ input size $n$ of instance = length of the *encoding* of instance

⇝ all running times are worst case over instances of encoding length $n$

# Karp Reductions


$\leq_p$


$A$: old problem
**Consensus: hard**

$B$: new problem
**Status: unknown**
(seems hard to *us* . . . )

- **Goal:** Show that $B$ is at least as hard as $A$.

short for: deterministic TM $M$ w/ $Time_M(n) = O(n^k)$ for constant $k$.

Assume there *was* a polytime algorithm $M$ für $B$.
Solve $A$ using $M$ (in polytime).

$\rightsquigarrow$ polytime algo for $B$ implies polytime algo for $A$

$\rightsquigarrow$ $B$ at least as hard as $A$

**Formally:** (strong notion than intuition above!)

## Definition 2.19 (polytime reduction, $\leq_p$)

Let $A \subseteq \Sigma^\star$ and $B \subseteq \Gamma^\star$ be languages (decision problems).
$A$ is *polytime reducible to* $B$ – written $A \leq_p B$ – if there is a total function $g : \Sigma^\star \to \Gamma^\star$, computable in polynomial time, with

$$\forall w \in \Sigma^\star \;:\; w \in A \Leftrightarrow g(w) \in B.$$

◀

- This type of reduction is called a *Karp-reduction*
- It is more restrictive than our intuitive version would need, but allows finer complexity classification (NP and co-NP)

# Implication of Reductions

## Lemma 2.20 (Membership reduction)

If $A \le_p B$ and $B \in \mathsf{P}$ (resp. $B \in \mathsf{NP}$), then $A \in \mathsf{P}$ (resp. $A \in \mathsf{NP}$).

**Proof:**

Since $B \in \mathsf{P}$, there is polytime TM $M$ with $\mathcal{L}(M) = B$.
Since $A \le_p B$, there further is polytime TM $g$ mit $\forall w : w \in A \Leftrightarrow g(w) \in B \quad (*)$.
We construct TM $M'$ for $A$:
first simulate $g$ on input $w$, then simulate $M$ on $g(w)$.

Since $M$ and $g$ are polytime TMs, so is $M'$, and $\mathcal{L}(M') = A$ (since $(*)$).
$\rightsquigarrow \quad A \in \mathsf{P}$.

(The version with $B \in \mathsf{NP}$ is similar, just using nondeterministic polytime). ■

# NP Completeness

## Definition 2.21 (NP-hard, NP-complete)

A language $A$ is called NP-*hard*, if we have for **all** languages $L \in \mathsf{NP}$ that $L \le_p A$.
A language $A$ is called NP-*complete*, if $A$ is NP-hard and $A \in \mathsf{NP}$.

## Theorem 2.22 (One for all and all for one)

For an NP-complete language $A$ holds: $A \in \mathsf{P} \iff \mathsf{P} = \mathsf{NP}$.

$\rightsquigarrow$ Under the *consensus hypothesis* that $\mathsf{P} \subsetneq \mathsf{NP}$, this means that
for an NP-complete problem $A$, we should expect **no efficient solution** for $A$.

**Proof:**

$\Rightarrow$ Let $L \in \mathsf{NP}$ be arbitrary.
Since $A$ is (by assumption) NP-hard, we have $L \le_p A$.
Since $A \in \mathsf{P}$, by membership reduction (Lemma **??**) also $L \in \mathsf{P}$.
Since $L$ was an *arbitrary* language from NP, $\mathsf{P} = \mathsf{NP}$ follows.

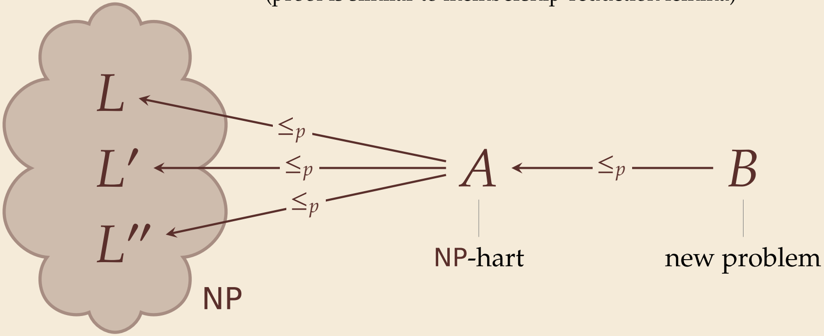
$\Leftarrow$ Let conversely $\mathsf{P} = \mathsf{NP}$.
Then, by assumption, $A \in \mathsf{NP} = \mathsf{P}$. ∎

# Implications of NP-Completeness

- NP-completeness tells us a lot about a problem!

But shall we possible prove $L \leq_p A$ *for all* possible $L \in \mathsf{NP}$?

- One can show: $\leq_p$ is a *transitive* relation on languages.
  (proof is similar to membership-reduction lemma)




$A$ NP-hard and $A \leq_p B$
$\rightsquigarrow$ $B$ NP-hard

# The Mother of All Problems

- It remains to identify a first NP-complete problem! Are there any *at all*?

### Theorem 2.23 (Cook-Levin)

SAT is NP-complete.

- SAT is the *satisfiability problem* of propositional logic:
  **Given:** Boolean (propositional logic) formula $\varphi$ over variables $x_1, \ldots, x_n$.
  **Goal:** Is there a variable assignment $V : \{x_1, \ldots, x_n\} \to \{\mathit{true}, \mathit{false}\}$,
  so that $\varphi$ evaluates under $V$ to $\mathit{true}$?

**Proof (Theorem ??):**

**Idea:** Given any nondeterministic TM $M$ for an arbitrary language $L \in \mathsf{NP}$, construct from a word $w$ a formula $\varphi(w)$, which exactly encodes all valid computations of $M$.

Variables $x_{q,t}$: Is $M$ be in state $q$ at time $t$?

$y_{c,i,t}$: Does tape cell $i$ contain char $c$ at time $t$?

$z_{i,t}$ Does read/write head stand at position $i$ at time $t$?

$\rightsquigarrow$ $\varphi(w)$ satisfiable iff $M$ accepts $w$.

(for details, see, e. g., §2.3 of S. Arora and B. Barak: *Computational Complexity: A Modern Approach*) ∎

## 2.7 Example of an NP-completeness proof

# 3SAT

- Let's do a more typical full example.
- Need one more NP-complete problem first

### Definition 2.24 (3SAT)

**Given:** A Boolean formula $\varphi$ in *3-CNF*:
*conjunctive normal form* with *at most* 3 literals per clause

**Goal:** Is there an assignment $V$ of the variables in $\varphi$, so that $\varphi$ evaluates to *true*?
(a. k. a. Is $\varphi$ *satisfiable*?) ◀

**Example:**

$(x_1 \vee \neg x_3 \vee x_2) \ \wedge \ (\neg x_3 \vee x_4 \vee \neg x_5) \ \wedge \ (x_5 \vee \neg x_5 \vee \neg x_1) \ \wedge \ (x_1 \vee x_3 \vee x_5)$

satisfiable, e. g., via $x_1 \mapsto \mathit{true}$, $x_5 \mapsto \mathit{false}$ (other variables arbitrary)

### Theorem 2.25 (3SAT)

SAT $\le_p$ 3SAT and 3SAT $\in$ NP. ◀

### Corollary 2.26 (3SAT)

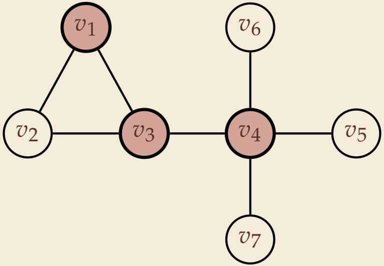
3SAT is NP-complete. ◀

# Vertex Cover

## Definition 2.27 (VertexCover)

**Given:** A (simple, undirected) graph $G = (V, E), E \subseteq \binom{V}{2}$, threshold $k$.

**Goal:** $\exists S \subseteq V$ with $|S| \le k$, such that $\forall e \in E : S \cap e \neq \emptyset$? ◀




Intuitively: a small subset $S$ of vertices of a graph, such that every edge is *covered* by $S$

## Theorem 2.28 (VertexCover hard)

VertexCover is NP-complete. ◀

**Proof:**

We will prove (i) VertexCover $\in$ VP and (ii) 3SAT $\le_p$ VertexCover.

$\rightsquigarrow$ Theorem **??** (since 3SAT is NP-complete and VP = NP).

(i) certificate = $S$; verifier whether all edges $e \in E$ are covered and whether $|S| \le k$

$\rightsquigarrow$ clearly doable in polytime $\rightsquigarrow$ VertexCover $\in$ VP.

# (ii) 3SAT $\leq_p$ VertexCover

**Proof:**

- Intuition: Express 3SAT instance as a VertexCover instance.
- So, let $\varphi$ be an arbitrary formula in 3-CNF over variables $x_1, \ldots, x_m$

$\rightsquigarrow$ $\varphi$ has the form $\underbrace{(l_{1,1} \vee l_{1,2} \vee l_{1,3})}_{C_1} \wedge \underbrace{(l_{2,1} \vee l_{2,2} \vee l_{2,3})}_{C_2} \wedge \cdots \wedge \underbrace{(l_{n,1} \vee l_{n,2} \vee l_{n,3})}_{C_n}$,
with $l_{i,j} \in \{x_1, \neg x_1, \ldots, x_m, \neg x_m\}$ for $i = 1, \ldots, n$ and $j = 1, 2, 3$.

- Define a graph $G = (V, E)$ via

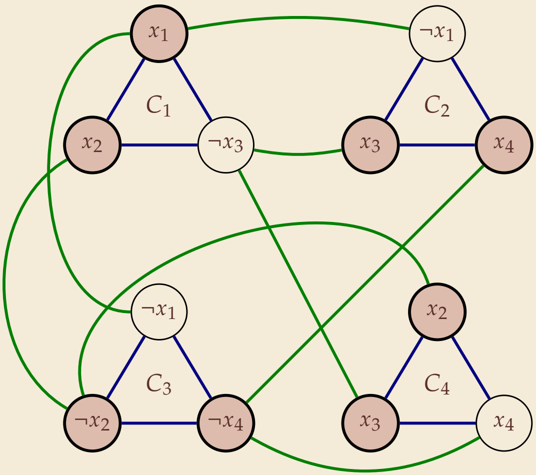
$$V = \{L_{i,j} : i = 1, \ldots, n;\ j = 1, 2, 3\}$$
$$E = \Big\{\{L_{i,j}, L_{p,q}\} : l_{i,j} \equiv \neg l_{p,q}\Big\} \cup \Big\{\{L_{i,1}, L_{i,2}\}, \{L_{i,2}, L_{i,3}\}, \{L_{i,3}, L_{i,1}\} : i = 1, \ldots, n\Big\}$$

We “draw” a vertex for every literal of a clause. We connect them if
(a) they are literals in the same clause or (b) they are negations of each other

$\rightsquigarrow$ **Claim:** $\varphi$ satisfiable $\iff$ $G$ has vertex cover of size $\leq 2n$.

# (ii) 3SAT $\le_p$ VertexCover – Example

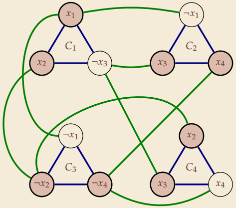
$\varphi = (x_1 \lor x_2 \lor \neg x_3) \land (\neg x_1 \lor x_3 \lor x_4) \land (\neg x_1 \lor \neg x_2 \lor \neg x_4) \land (x_2 \lor x_3 \lor x_4)$




Set $S$ (a VC of size $2n$)

- **Idea:** Vertices ***not in*** vertex cover $S$ define a variable assignment.
  - Cannot be contradictory, otherwise "negation"-edge not covered.
  - Must take $\ge 2$ vertices per clause into $S$ (otherwise triangle not covered)
  - $\rightsquigarrow$ $|S| \ge 2n$ for every vertex cover.
- In the example:
  - Fat vertices form a vertex cover for $G$
  - corresponding assignment: $V = \{x_1 \mapsto 0, x_2 \mapsto 0, x_3 \mapsto 0, x_4 \mapsto 1\}$ $(0 \mathrel{\widehat{=}} \mathit{false}, 1 \mathrel{\widehat{=}} \mathit{true})$
  - $\rightsquigarrow$ $\varphi$ satisfiable

# (ii) 3SAT $\leq_p$ VertexCover – Correctness Proof




**Claim:** $\varphi$ satisfiable $\iff$ $G$ has VC $S$ of size $|S| \leq 2n$.

$\Rightarrow$ Let $\varphi$ erfüllbar; $\rightsquigarrow$ every $C_i$ has a satisfied literal $l_{i,j}$.
Add the *other two* vertices of the clause to $S$.
$\rightsquigarrow$ $|S| = 2n$ and $S$ covers all clause triangle edges.
Remaining edges have the form $\{x, \neg x\}$.
If such an edge remained uncovered by $S$, we would have that both $x$ and $\neg x$ are satisfied literals $\lightning$
$\rightsquigarrow$ $G$ has VC of size $2n$.

$\Leftarrow$ Given a VC $S$ of $G$ with $|S| \leq 2n$.
$S$ must contain 2 vertices per clause triangle $\rightsquigarrow$ $|S| = 2n$ and $S$ is a *minimal* VC.
Define assignment $V$ so that all literals *not* in $S$ are satisfied.
(Variables which are not assigned a value via this procedure can be assigned an arbitrary value.)
$V$ is well defined, since $\{x, \neg x\}$-edges must be covered.
Moreover, $V$ makes $\varphi$ *true*:
from every clause, at least one literal is satisfied since $|S| = 2n$.
$\rightsquigarrow$ $\varphi$ satisfiable. ■

# (ii) 3SAT $\leq_p$ VertexCover – Running Time

- Construction of $G$ upon input $\varphi$ can easily be done in polytime
  - $|V| = O(n)$, $|E| = O(n^2)$
  - Construction of $E$ in time $O(n^2)$ easy to do, e. g., on RAM $\rightsquigarrow$ $\exists$ polytime TM.

$\rightsquigarrow$ 3SAT $\leq_p$ VertexCover. ∎

## 2.8 Important NP-Complete Problems

# Further NP-complete problems [1]

Apart from SAT, 3SAT, and VertexCover, here are some of the most useful NP-complete problems.

## Definition 2.29 (Dominating Set)

Given: graph $G = (V, E)$ and $k \in \mathbb{N}$
Question: $\exists V' \subset V \quad : \quad |V'| \leq k \ \wedge \ \forall v \in V : \big(v \in V' \ \vee \ \exists u \in N(v) : u \in V'\big)$ ◀

## Definition 2.30 (Hamiltonian Cycle)

Given: graph $G = (V, E)$ (directed and undirected version)
Question: Is there a vertex-simple cycle in $G$ of length $|V|$? ◀

## Definition 2.31 (Clique)

Given: graph $G = (V, E)$ and $k \in \mathbb{N}$
Question: $\exists V' \subset V \quad : \quad |V'| \geq k \ \wedge \ \forall u, v \in V' : \{u, v\} \in E$ ◀

## Definition 2.32 (Independent Set)

Given: graph $G = (V, E)$ and $k \in \mathbb{N}$
Question: $\exists V' \subset V \quad : \quad |V'| \geq k \ \wedge \ \forall u, v \in V' : \{u, v\} \notin E$ ◀

# Further NP-complete problems [2]

## Definition 2.33 (Traveling Salesperson (TSP))

Given: distance matrix $D \in \mathbb{N}^{n \times n}$ and $k \in \mathbb{N}$

Question: Is there a permutation $\pi : [n] \to [n]$ with $\sum_{i=1}^{n-1} D_{\pi(i),\pi(i+1)} + D_{\pi(n),\pi(1)} \leq k$ ? ◀

## Definition 2.34 (Graph Coloring)

Given: graph $G = (V, E)$ and $k \in \mathbb{N}$

Question: $\exists c : V \to [k] \quad : \quad \forall \{u, v\} \in E : c(u) \neq c(v)$ ? ◀

## Definition 2.35 (Set Cover)

Given: $n \in \mathbb{N}$, sets $S_1, \ldots, S_m \subseteq [n]$ and $k \in \mathbb{N}$

Question: $\exists I \subseteq [m] \quad : \quad \bigcup_{i \in I} S_i = [n] \ \wedge \ |I| \leq k$ ? ◀

## Definition 2.36 (Weighted Set Cover)

Given: $n \in \mathbb{N}$, sets $S_1, \ldots, S_m \subseteq [n]$, costs $c_1, \ldots, c_m \in \mathbb{N}_0$ and $k \in \mathbb{N}$

Question: $\exists I \subseteq [m] \quad : \quad \bigcup_{i \in I} S_i = [n] \ \wedge \ \sum_{i \in I} c_i \leq k$ ? ◀

# Further hard problems [3]

### Definition 2.37 (Closest String)

Given: $s_1, \ldots, s_n \in \Sigma^m$ and $k \in \mathbb{N}$

Question: $\exists s \in \Sigma^m \quad : \quad \forall i \in [n] : d_H(s, s_i) \leq k$ ?   ($d_H$ Hamming-distance) ◀

### Definition 2.38 (Max Cut)

Given: graph $G = (V, E)$ and $k \in \mathbb{N}$

Question: $\exists C \subset V \quad : \quad \big|E \cap \{\{u, v\} \mid u \in C, v \notin C\}\big| \geq k$ ? ◀

### Definition 2.39 (Exact Cover)

Given: $n \in \mathbb{N}$, sets $S_1, \ldots, S_m \subseteq [n]$

Question: $\exists I \subseteq [m] \quad : \quad \bigcup_{i \in I} S_i \;=\; [n] \;\wedge\; \sum_{i \in I} |S_i| = n$ ? ◀

# Further hard problems [4]

## Definition 2.40 (Subset Sum)

Given: $x_1, \ldots, x_n \in \mathbb{Z}$

Question: $\exists I \subseteq [n] \ : \ I \neq \emptyset \wedge \sum_{i \in I} x_i = 0$ ? ◀

## Definition 2.41 ((0/1) Knapsack)

Given: $w_1, \ldots, w_n \in \mathbb{N}$, $v_1, \ldots, v_n \in \mathbb{N}$ and $b, k \in \mathbb{N}$

Question: $\exists I \subseteq [n] \ : \ \sum_{i \in I} w_i \leq b \ \wedge \ \sum_{i \in I} v_i \geq k$ ? ◀

## Definition 2.42 (Bin Packing)

Given: $w_1, \ldots, w_n \in \mathbb{N}$, $b \in \mathbb{N}$, $k \in \mathbb{N}$

Question: $\exists a : [n] \to [k] \ : \ \forall j \in [k] : \displaystyle\sum_{\substack{i=1,\ldots,n \\ a[i]=j}} w_i \ \leq \ b$ ? ◀

## Definition 2.43 (0/1 Integer Programming)

Given: integer linear program (ILP) $A \in \mathbb{Z}^{m \times n}$, $b \in \mathbb{Z}^m$ and $c \in \mathbb{Z}^n$ and $k \in \mathbb{Z}$

Question: Is there $x \in \{0,1\}^n$ with $Ax \leq b$ and $c^T x \geq k$ ? ◀

## 2.9 Optimization Problems

# Optimization Problems

## Definition 2.44 (Optimization Problem)

An *optimization problem* is given by 7-tuple $U = (\Sigma_I, \Sigma_O, L, L_I, M, cost, goal)$ with

1. $\Sigma_I$ an alphabet (called input alphabet),
2. $\Sigma_O$ an alphabet (called output alphabet),
3. $L \subseteq \Sigma_I^\star$ the language of allowable problem instances (for which $U$ is well-defined),
4. $L_I \subseteq L$ the language of actual problem instances for $U$ (for those we want to determine $U$'s complexity),
5. $M : L \to 2^{\Sigma_O^\star}$ and with $x \in L$, $M(x)$ is the set of all feasible solutions for $x$.
6. $cost$ is a cost function, which assigns for $x \in L$ each pair $(u, x)$ with $u \in M(x)$ a positive real number,
7. $goal \in \{\min, \max\}$.

◀

# Optimal Solutions

## Definition 2.45 (Optimal Solutions, Solution Algorithms)

Let $U = (\Sigma_I, \Sigma_O, L, L_I, M, cost, goal)$ an optimization problem. For each $x \in L_I$ a feasible solution $y \in M(x)$ is called *optimal for $x$ and $U$*, if

$$cost(y, x) \quad = \quad goal\{cost(z, x) \mid z \in M(x)\}.$$

◀

An algorithm $A$ is *consistent with $U$* if $A(x) \in M(x)$ for all $x \in L_I$.

We say *algorithm $B$ solves $U$*, if

1. $B$ is consistent with $U$ and
2. for all $x \in L_I$, $B(x)$ is optimal for $x$ and $U$.

◀

# Optimization Problems – Examples

Natural examples: Problems above with an input parameter $k$.

Less immediate example:

**Definition 2.46 (Max-SAT)**

Given: CNF-Formula $\phi = C_1 \wedge \cdots \wedge C_m$ over variables $x_1, \ldots, x_n$
Allowable (=Actual) Instances: encodings of $\phi$
$M(\phi) = \{0, 1\}^n$ (variable assignments)
$cost(u, x)$: # of satisfied clauses in $u$ under given assignment $x$
$goal = \max$ ◀

# Classes of Optimization Problems

## Definition 2.47 (NPO)

NPO is the class if optimization problems $U = (\Sigma_I, \Sigma_O, L, L_I, M, cost, goal)$ with

1. $L_I \in \mathsf{P}$,
2. there is a polynomial $p_U$ with
   a) $\forall x \in L_I\ \forall y \in M(x) : |y| \leq p_U(|x|)$ and
   b) there is a polynomial time algorithm which for all $y \in \Sigma_O^\star$, $x \in L_I$ with $|y| \leq p_U(|x|)$ decides whether $y \in M(x)$ holds, and
3. function $cost$ can be computed in polynomial time. ◀

## Definition 2.48 (PO)

PO is the class of optimization problems $U = (\Sigma_I, \Sigma_O, L, L_I, M, cost, goal)$ with

1. $U \in$ NPO, and
2. there is an algorithm of polynomial time complexity which for all $x \in L_I$ computes an optimal solution for $x$ and $U$. ◀

# Glossary of Problem Types

# From Optimization to Decision

## Definition 2.49 (Threshold Languages)

Let $U = (\Sigma_I, \Sigma_O, L, L_I, M, cost, goal)$ an optimization problem, $U \in \mathsf{NPO}$.
For $Opt_U(x)$ the cost of an optimal solutions for $x$ and $U$ we define the *threshold language for U* as

$$Lang_U = \begin{cases} \left\{(x,k) \in L_I \times \{0,1\}^\star \;\middle|\; Opt_U(x) \le k_2\right\}, & \text{if } goal = \min, \\ \left\{(x,k) \in L_I \times \{0,1\}^\star \;\middle|\; Opt_U(x) \ge k_2\right\}, & \text{if } goal = \max. \end{cases}$$

We say $U$ *is* NP-*hard*, if $Lang_U$ is NP-hard. ◀

# Hardness of Optimization

## Corollary 2.50 (Optimization is harder than Threshold)

Let $U$ an optimization problem.
If $Lang_U$ is NP-hard and if $\mathsf{P} \neq \mathsf{NP}$ holds, we have $U \notin \mathsf{PO}$.

**Proof:**

Contraposition. Suppose $U \in \mathsf{PO}$.

$\rightsquigarrow$ $\exists$ polytime algo $A$ that computes optimal $y$ for any instance $x$ of $U$
$\rightsquigarrow$ can compute $Opt_U(x) = cost(y)$ in polytime
$\rightsquigarrow$ can decide whether $Opt_U(x)$ better than threshold in polytime
$\rightsquigarrow$ $Lang_U \in \mathsf{P}$.

↯ $Lang_U$ NP-hard $\rightsquigarrow$ $U \notin \mathsf{PO}$. ∎

# Max-SAT is hard

## Corollary 2.51 (Max-SAT is hard)

MAX-SAT is NP-hard.

**Proof:**

We show: $3\mathrm{SAT} \leq_p \mathit{Lang}_{\text{MAX-SAT}}$.

Given a 3SAT instance $x$ encoding a formula with $m$ clauses, output $(x, m)$.

$x \in 3\mathrm{SAT} \iff x$ is satisfiable $\iff m$ clauses of $x$ satisfiable $\iff (x, m) \in \mathit{Lang}_{\text{MAX-SAT}}$. ∎

# Summary

- We have formalized the classic notion of intractable problems.
  - What is running time, what is "polytime"?
  - Decision problems $\leftrightarrow$ (formal) languages
  - P, NP via Turing machines $\leftrightarrow$ certificates and verifiers
  - For the typical case of optimization problems,
    there are different versions of the problem,
    but (in)tractability typically carries over.

$\rightsquigarrow$ We can mathematically prove a problem is intractable (NP-hard).

. . . but how can we tackle hard problems anyway?