

# COMPLEXITY THEORY

## Lecture 5: Time Complexity and Polynomial Time

Markus Krötzsch

Knowledge-Based Systems

TU Dresden, 26th Oct 2021

More recent versions of this slide deck might be available.  
For the most current version of this course, see  
[https://iccl.inf.tu-dresden.de/web/Complexity\\_Theory](https://iccl.inf.tu-dresden.de/web/Complexity_Theory)

# Time Complexity

# Measuring Complexity

## Complexity Theory

Study the fine structure of decidable languages.

## Goal

Classify languages by the amount of resources needed to solve them.

## Resources

When dealing with Turing machines, we will primarily consider

- **time**: the running time of algorithms (steps on a Turing-machine)
- **space**: the amount of additional memory needed  
(cells on the Turing-tapes)

# Time and Space Bounded Turing Machines

**Definition 5.1:** Consider a Turing machine  $\mathcal{M}$  and a function  $f : \mathbb{N} \rightarrow \mathbb{R}^+$ .

- (1)  $\mathcal{M}$  is  **$f$ -time bounded** if it halts on every input  $w \in \Sigma^*$  after  $\leq f(|w|)$  steps.
- (2)  $\mathcal{M}$  is  **$f$ -space bounded** if it halts on every input  $w \in \Sigma^*$  using  $\leq f(|w|)$  cells on its tapes.

(Here we typically assume that Turing machines have a separate input tape that we do not count in measuring space complexity.)

**Notation 5.2:** Sometimes notations like “ $f(n)$ -time bounded” are used, assuming inputs to be of length  $n$

$\leadsto$  we use this when convenient, e.g., to write “ $n^3$ -bounded”

# Big-O and Small-o

Algorithms are often judged by their asymptotic complexity, i.e., their behaviour in the limit.

We recall and extend the definition from Lecture 1:

**Definition 5.3:** The **Big-O notation** classifies functions using asymptotic upper bounds:

$$f(n) = O(g(n)) \quad \text{iff} \quad \exists c > 0 \exists n_0 \in \mathbb{N} \forall n > n_0: f(n) \leq c \cdot g(n)$$

Then  $f$  is **asymptotically bounded** by  $g$  up to a constant factor.

**Definition 5.4:** The **small-o notation** classifies by a function that **dominates** them:

$$f(n) = o(g(n)) \quad \text{iff} \quad \forall c > 0 \exists n_0 \in \mathbb{N} \forall n > n_0: f(n) \leq c \cdot g(n)$$

Then  $f$  is **asymptotically dominated** by  $g$ .

# Relatives of the $O$ Notation

There are a number of further asymptotic notations besides Big-O and small-o. Their essence and underlying intuition is as follows:

Notation	$C = \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$	Intuition
$f \in O(g)$	$C < \infty$	" $f \leq g$ "
$f \in \Omega(g)$	$C > 0$	" $f \geq g$ "
$f \in \Theta(g)$	$0 < C < \infty$	" $f = g$ "
$f \in o(g)$	$C = 0$	" $f < g$ "
$f \in \omega(g)$	$C = \infty$	" $f > g$ "

**Note:** Both " $f \in O(g)$ " and " $f = O(g)$ " etc. are sometimes used in the literature, with the same intended meaning.

# Relaxed Time and Space Bounds

We can use Big-O notation to generalise bounded TMs:

**Definition 5.5:** A Turing machine  $\mathcal{M}$  is

- (1)  $O(g(n))$ -time bounded if it is  $f$ -time bounded for some  $f$  with  $f(n) = O(g(n))$
- (2)  $O(g(n))$ -space bounded if it is  $f$ -space bounded for some  $f$  with  $f(n) = O(g(n))$

**Notation 5.6:** We generally allow the use of  $O(g(n))$  in place of a function  $f(n)$  with analogous meaning.

# Deterministic Complexity Classes

Bounding TMs is the basis for both complexity theory and for studies of algorithmic complexity.

**Definition 5.7:** Let  $f : \mathbb{N} \rightarrow \mathbb{R}^+$  be a function.

- (1) **DTime**( $f(n)$ ) is the class of all languages  $\mathbf{L}$  for which there is an  $O(f(n))$ -time bounded deterministic Turing machine deciding  $\mathbf{L}$ .
- (2) **DSpace**( $f(n)$ ) is the class of all languages  $\mathbf{L}$  for which there is an  $O(f(n))$ -space bounded deterministic Turing machine deciding  $\mathbf{L}$ .

**Notation 5.8:** Sometimes  $\text{Time}(f(n))$  is used instead of  $\text{DTime}(f(n))$ .



# Is Complexity Theory Impossible in Practice?

The classes  $\text{DTIME}(f)$  and  $\text{DSPACE}(f)$  depend on

- details of the computational model
- details of the input encoding
- details of the implementation

An exact specification of such bounds is often extremely hard.

**Example 5.9:** A naive algorithm can perform matrix multiplication in  $\text{DTIME}(n^3)$ .

Since many decades, researchers have been searching for better solutions:  
 $\text{DTIME}(n^{2.808})$  [Strassen, 1969],  $\text{DTIME}(n^{2.796})$  [Pan, 1978],  $\text{DTIME}(n^{2.780})$  [Bini et al., 1979],  $\text{DTIME}(n^{2.522})$  [Schönhage, 1981],  $\text{DTIME}(n^{2.517})$  [Romani, 1982],  $\text{DTIME}(n^{2.496})$  [Coppersmith & Winograd, 1981],  $\text{DTIME}(n^{2.479})$  [Strassen, 1986],  $\text{DTIME}(n^{2.374})$  [Coppersmith & Winograd, 1990],  $\text{DTIME}(n^{2.374})$  [Stothers, 2010],  $\text{DTIME}(n^{2.373})$  [Williams, 2011], and  $\text{DTIME}(n^{2.37286})$  [Alman & Williams, 2020]

# Defining Complexity Classes

**Solution:** Make complexity classes big enough to hide such details.

$$P = PTime = \bigcup_{d \geq 1} DTime(n^d) \quad \text{polynomial time}$$

$$Exp = ExpTime = \bigcup_{d \geq 1} DTime(2^{n^d}) \quad \text{exponential time}$$

$$2Exp = 2ExpTime = \bigcup_{d \geq 1} DTime(2^{2^{n^d}}) \quad \text{double-exponential time}$$

$$E = ETime = \bigcup_{d \geq 1} DTime(2^{dn}) \quad \text{exp. time with linear exponent}$$

$$L = LogSpace = DSpace(\log n) \quad \text{logarithmic space}$$

$$PSpace = \bigcup_{d \geq 1} DSpace(n^d) \quad \text{polynomial space}$$

$$ExpSpace = \bigcup_{d \geq 1} DSpace(2^{n^d}) \quad \text{exponential space}$$

# Time Complexity Classes

$$P = PTime = \bigcup_{d \geq 1} DTime(n^d) \quad \text{polynomial time}$$

$$Exp = ExpTime = \bigcup_{d \geq 1} DTime(2^{n^d}) \quad \text{exponential time}$$

$$2Exp = 2ExpTime = \bigcup_{d \geq 1} DTime(2^{2^{n^d}}) \quad \text{double-exponential time}$$

**Note:** Complexity classes are classes of languages.

**Observation:** The following relationships are clear from the definition:

$$P \subseteq ExpTime \subseteq 2ExpTime \subseteq 3ExpTime \subseteq 4ExpTime \subseteq \dots$$

# A Hierarchy of Complexity Classes?

Many fundamental questions arise:

- Can we always solve more problems if we have more resources?
- If not, how much more resources do we need to be able to solve strictly more problems?
- How do the complexity classes relate to each other?
- Are there any tools by which we can show that a problem is in any of these classes but not in another?

~> discussed in future lectures

- How do we classify “efficient” in terms of complexity classes?

~> coming up next

# Different Definitions of Complexity Classes?

How is complexity affected by the chosen model of computation?

- Is  $DTime(f)$  the same for multi-tape TMs?
- And how about non-deterministic TMs?
- Or TMs with a two-way infinite tape?
- Or random access machines?
- ...

Many complexity classes are **robust** against many such variations

↪ coming up next

# Polynomial Time

# Polynomial Time

An “intuitive” definition of “efficient”:

- Any linear time computation is “efficient”.
- Any program that
  - performs “efficient” operations (e.g. linear number of iterations) and
  - only uses “efficient” subprogramsis “efficient”.

This turns out to be equivalent to PTime.

$$\text{PTime} := \bigcup_{d \geq 1} \text{DTime}(n^d)$$

PTime serves as a mathematical model of “efficient” computation.

# Robustness of the Definition

If PTime is to be the mathematical model of efficient computation, it should not depend on

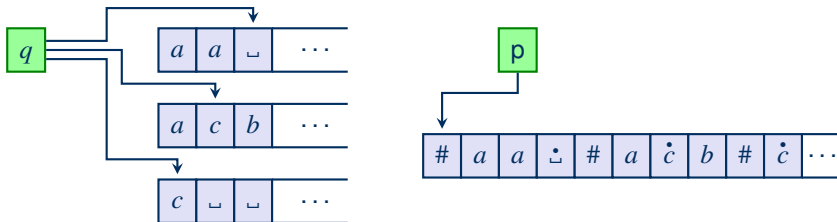
- the exact computational model we are using,
- or how we encode the input (within reason).



# Multi-Tape Turing Machines

**Theorem 5.10 (Sipser, Theorem 7.8):** Consider a function  $f$  with  $f(n) \geq n$ . Then, for every  $f(n)$ -time bounded  $k$ -tape Turing machine ( $k > 1$ ), there is an equivalent  $O(f^2(n))$ -time bounded single-tape Turing machine.

**Proof:** Simulate a multi-tape TM with a single-tape TM as shown in Lecture 2:



# Multi-Tape Turing Machines

**Theorem 5.10 (Sipser, Theorem 7.8):** Consider a function  $f$  with  $f(n) \geq n$ . Then, for every  $f(n)$ -time bounded  $k$ -tape Turing machine ( $k > 1$ ), there is an equivalent  $O(f^2(n))$ -time bounded single-tape Turing machine.

**Proof (cont.):** Then analyse how long this simulation really takes:

- **Observation:** the tapes can never have more than  $f(n)$  symbols on them
- The simulation scans the whole tape once to find out what to do:  
 $O(f(n))$  steps
- Then it updates the whole tape in one pass:  $O(f(n))$  steps
- Sometimes the whole tape is shifted to make space:  
at most  $k$  times  $O(f(n))$  steps
- Overall: one step is simulated in  $O(f(n))$  steps
- Simulating  $f(n)$  such steps takes  $f(n) \cdot O(f(n)) = O(f^2(n))$  steps
- Tape initialisation takes another  $O(f(n))$  (irrelevant)

Total simulation possible in  $O(f^2(n))$ .

□

# P is Robust for Multi-Tape TMs

Let  $\text{DTime}_k(f(n))$  denote “ $\text{DTime}(f(n))$  for a  $k$ -tape TM”.

## Theorem 5.11:

$$\bigcup_{d \in \mathbb{N}} \text{DTime}(n^d) = \bigcup_{d \in \mathbb{N}} \text{DTime}_k(n^d) \text{ for every } k \geq 1$$

**Proof:** The inclusion  $\subseteq$  is clear.

The inclusion  $\supseteq$  follows from the previous Theorem 5.10. □

# Robustness Against Other Models of Computation

P is robust against further models of computation:

- (1) We can simulate  $f(n)$  steps of a two-way infinite  $k$ -tape Turing-machine with an equivalent standard  $k$ -tape TM in  $O(f(n))$  steps.
- (2) We can simulate  $f(n)$  steps of a RAM-machine with a 3-tape TM in  $O(f^3(n))$  steps. Vice-versa in  $O(f(n))$  steps.

Consequences:

- PTime is the same for all these models (unlike linear time)
- The exponential time complexity classes are as robust as P

How about non-deterministic TMs?

It is unknown if PTime is robust against this, but most think it is not

↪ see next lectures

# Linear Speed-Up

The Big-O notation in DTime hides arbitrary linear factors.

Is it justified to rely on this for defining P?

Yes, it turns out that we can make multi-tape TMs “arbitrarily fast”:

**Theorem 5.12 (Linear Speed-Up Theorem):** Consider an  $f(n)$ -time bounded  $k$ -tape Turing machine  $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$  with  $k > 1$ .

Then, for every constant  $c > 0$ , there is a  $(\frac{1}{c} \cdot f(n) + n + 2)$ -time bounded  $k$ -tape TM  $\mathcal{M}' = (Q', \Sigma, \Gamma', \delta', q'_0, q'_{\text{accept}}, q'_{\text{reject}})$  that accepts the same language.

# Linear Speed-Up (Proof)

**Proof (sketch):** Let  $\Gamma' := \Sigma \cup \Gamma^m$  where  $m := \lceil 6c \rceil$ . We construct  $\mathcal{M}'$  as follows:

Step 1: Compress  $\mathcal{M}$ 's input.

Copy the input to tape 2, compressing  $m$  symbols into one (i.e., each symbol corresponds to an  $m$ -tuple from  $\Gamma^m$ ). This takes  $n + 2$  steps.

Step 2: Simulate  $\mathcal{M}$ 's computation,  $m$  steps at once.

- (1) Read (in 4 steps) symbols to the left, right and the current position and “store” in  $Q'$ , using  $|Q \times \{1, \dots, m\}^k \times \Gamma^{3mk}|$  extra states.
- (2) Simulate (in 2 steps) the next  $m$  steps of  $\mathcal{M}$  (as  $\mathcal{M}$  can only modify the current position and one of its neighbours)
- (3)  $\mathcal{M}'$  accepts (rejects) if  $\mathcal{M}$  accepts (rejects)

For further details see Papadimitriou, Theorem 2.2.

□

# Different Encodings

Some simple observations:

- (1) For any  $n \in \mathbb{N}$ , the length of the encoding of  $n$  in base  $b_1$  and base  $b_2$  are related by a constant factor, for all  $b_1, b_2 \geq 2$ .
- (2) For any graph  $G$ , the length of its encoding as an
  - adjacency matrix
  - list of nodes + list of edges
  - adjacency list
  - ...

are all polynomially related.

Consequence:

PTime is the same for all these encodings (unlike linear time).

# PTime = tractable?

The class PTime is a reasonable mathematical model of the class of problems which are tractable or solvable in practice.

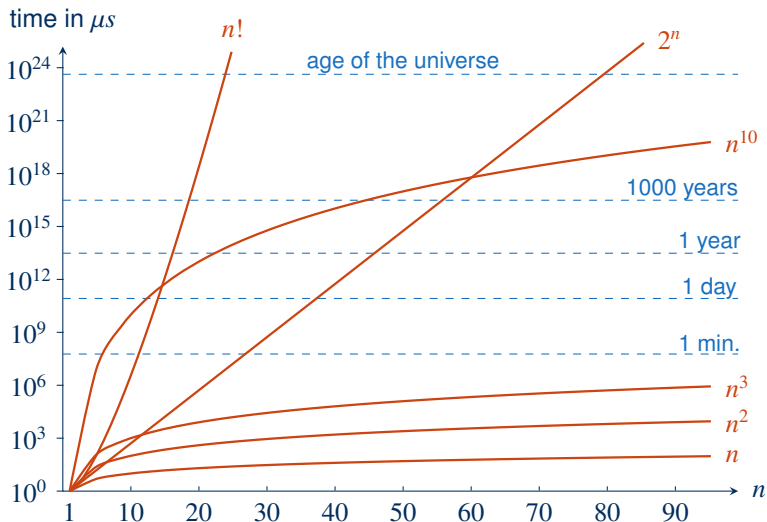
However: This correspondence is not exact.

- When the degree of polynomials is very high, the time grows so quickly that in practice the problem is not solvable.
- The constants may also be very large

And yet: For many concrete PTime-problems arising in practice, algorithms with moderate exponents and constants have been found.



# Growth Rate of Some Functions



# Problems in P

# Proving a Problem is in PTime

- The most direct way to show that a problem is in PTime is to exhibit a polynomial time algorithm that solves it.
- Even a naive polynomial-time algorithm often provides a good insight into how the problem can be solved efficiently.
- Because of robustness, we do not generally need to specify all the details of the machine model or the encoding.

~> pseudo-code is sufficient

# Example: Satisfiability

Some of the most important problems concern logical formulae

**Definition 5.13 (Propositional Logic Syntax):** Formulae of **propositional logic** are built up inductively

- (Propositional) Variables:  $X_i \quad i \in \mathbb{N}$
- Boolean connectives: If  $\varphi, \psi$  are propositional formulae then so are
  - $(\psi \vee \varphi)$
  - $(\psi \wedge \varphi)$
  - $\neg\varphi$

**Example 5.14:** The following is a propositional logic formula:

$$(X_1 \vee X_2 \vee \neg X_5) \wedge (\neg X_2 \vee \neg X_4 \vee \neg X_5) \wedge (X_2 \vee X_3 \vee X_4)$$

# Conjunctive Normal Form

**Definition 5.15 (Conjunctive Normal Form):** A propositional logic formula  $\varphi$  is in **conjunctive normal form** (CNF) if

$$\varphi = C_1 \wedge \cdots \wedge C_m$$

where each  $C_i$  is a **clause**, that is, a disjunction of **literals**

$$C_i = (L_{i1} \vee \cdots \vee L_{ik})$$

and a **literal** is a variable  $X_i$  or a negation  $\neg X_i$  thereof.

A CNF  $\varphi$  is in  **$k$ -CNF** if it has at most  $k$  literals per clause.

**Example 5.16:** The following formula is in 3-CNF:

$$(X_1 \vee X_2 \vee \neg X_5) \wedge (\neg X_2 \vee \neg X_4 \vee \neg X_5) \wedge (X_2 \vee X_3 \vee X_4)$$

# Propositional Logic Semantics

**Definition 5.17:** A formula  $\varphi$  is **satisfiable** if it is satisfied by an assignment that maps each variable in  $\varphi$  to either 0 or 1 (and recursively defined for larger formulae as usual).

Specifically: A formula in CNF is satisfiable if there is an assignment  $\beta$  for variables of  $\varphi$  so that every clause contains at least

- one variable to which  $\beta$  assigns 1, or
- one negated variable to which  $\beta$  assigns 0.

**Example 5.18:** The formula

$$(X_1 \vee X_2 \vee \neg X_5) \wedge (\neg X_2 \vee \neg X_4 \vee \neg X_5) \wedge (X_2 \vee X_3 \vee X_4)$$

is satisfied by  $\{X_1 \mapsto 1, X_2 \mapsto 0, X_3 \mapsto 1, X_4 \mapsto 0, X_5 \mapsto 1\}$ .

# The Satisfiability Problem

Related to propositional formulae, the following two problems are the most important:

## **SAT**

Input: Propositional formula  $\varphi$  in CNF

Problem: Is  $\varphi$  satisfiable?

## ***k*-SAT**

Input: Propositional formula  $\varphi$  in  $k$ -CNF

Problem: Is  $\varphi$  satisfiable?

# 2-Sat is Polynomial

**Theorem 5.19:** 2-Sat  $\in$  PTime.

**Proof:** The following algorithm solves the problem in polynomial time.

**Main:** Input  $\Gamma$  in CNF

$\text{bcp}(\Gamma)$

**if** conflict **return** UNSAT

**while**  $\Gamma \neq \emptyset$  **do**

    choose var.  $X$  from  $\Gamma$

    set  $\Gamma' := \Gamma$

    assign( $\Gamma, X, 1$ )

$\text{bcp}(\Gamma)$

**if** conflict

$\Gamma := \Gamma'$

        assign( $\Gamma, X, 0$ )

$\text{bcp}(\Gamma)$

**if** conflict

**return** UNSAT

**return** SAT

$\text{bcp}(\Gamma)$  (boolean constraint propagation)

**while**  $\Gamma$  contains unit-clause  $C$  **do**

**if**  $C = \{X\}$       assign( $\Gamma, X, 1$ )

**if**  $C = \{\neg X\}$     assign( $\Gamma, X, 0$ )

**if**  $\Gamma$  contains empty clause **return** conflict

assign( $\Gamma, X, c$ )

**if**  $c = 1$

    remove from  $\Gamma$  all clauses  $C$  with  $X \in C$

    remove  $\neg X$  from all remaining clauses

**if**  $c = 0$

    remove from  $\Gamma$  all clauses  $C$  with  $\neg X \in C$

    remove  $X$  from all remaining clauses



# Polynomial-Time Reductions

As for decidability we can use reductions to show membership in PTime.

**Definition 5.20:** A language  $L_1 \subseteq \Sigma^*$  is **polynomially many-one reducible** to  $L_2 \subseteq \Sigma^*$ , denoted  $L_1 \leq_p L_2$ , if there is a polynomial-time computable function  $f$  such that for all  $w \in \Sigma^*$

$$w \in L_1 \quad \text{if and only if} \quad f(w) \in L_2.$$

**Theorem 5.21:** If  $L_1 \leq_p L_2$  and  $L_2 \in \text{PTime}$  then  $L_1 \in \text{PTime}$ .

**Proof:** The sum and composition of polynomials is a polynomial. □

# Reductions in PTime

All non-trivial members of PTime can be reduced to each other:

**Theorem 5.22:** If  $\mathbf{B}$  is any language in P,  $\mathbf{B} \neq \emptyset$ , and  $\mathbf{B} \neq \Sigma^*$ , then  $\mathbf{A} \leq_p \mathbf{B}$  for any  $\mathbf{A} \in \text{P}$ .

**Proof:** Choose  $w \in \mathbf{B}$  and  $w' \notin \mathbf{B}$ .

Define the function  $f$  by setting

$$f(x) := \begin{cases} w & \text{if } x \in \mathbf{A} \\ w' & \text{if } x \notin \mathbf{A} \end{cases}$$

Since  $\mathbf{A} \in \text{P}$ , this function  $f$  is computable in polynomial time, and it is a reduction from  $\mathbf{A}$  to  $\mathbf{B}$ . □

## Example: Colourability

**Definition 5.23 (Vertex Colouring):** A **vertex colouring** of  $G$  with  $k$  colours is a function

$$c : V(G) \longrightarrow \{1, \dots, k\}$$

such that adjacent nodes have different colours, that is:

$$\{u, v\} \in E(G) \text{ implies } c(u) \neq c(v)$$

### **$k$ -COLOURING**

Input: Graph  $G$ ,  $k \in \mathbb{N}$

Problem: Does  $G$  have a vertex colouring with  $k$  colours?

For  $k = 2$  this is the same as **BIPARTITE**.

# Reducing 2-Colourability to 2-Sat

**Theorem 5.24:**  $2\text{-COLOURABILITY} \leq_p 2\text{-SAT}$ , and therefore  $2\text{-COLOURABILITY} \in P$ .

**Proof:** We define a reduction as follows:      Given graph  $G$

- For each vertex  $v \in V(G)$  of the graph introduce new variable  $X_v$
- For each  $\{u, v\} \in E(G)$  add clauses  $(X_u \vee X_v)$  and  $(\neg X_u \vee \neg X_v)$

This is obviously computable in polynomial time.

We check that it is a reduction:

- If  $G$  is 2-colourable, use colouring to assign truth values.  
(One colour is true, the other false)
- If the formula is satisfiable, the truth assignment defines valid 2-colouring.  
For every edge  $\{u, v\} \in E(G)$ , one variable  $X_u, X_v$  must be set to true, the other to false.

□

# Trivially Tractable Problems

A large class of languages is generally tractable:

**Theorem 5.25:** If  $L$  is a finite language, then it is decided by an  $O(1)$ -time bounded TM. In other words, all finite languages are decidable in constant time (and hence also in polynomial time).

## Proof:

- As  $L$  is finite, there is a maximum length  $m$  of words in  $L$ .
- Read the input up to the first  $m$  letters.
- The state space contains a table containing the correct result for all such inputs.
- All other inputs are rejected. □

# A Note on Constructiveness

The next result is an example of a theorem that proves the existence of a P algorithm in cases where we do not know what this algorithm is.

**Example 5.26:** Let  $L$  be the language that contains all correct sentences from the following set:

{“P is the same as NP”, “P is not the same as NP”}

Then  $L$  is decidable in constant time.

However, we don't know which constant-time algorithm decides it.

Non-constructiveness:

- We can prove that there is a correct polynomial time algorithm.
- We cannot construct such an algorithm.

Such solutions are called **non-constructive**.

# An Interesting Problem in P

**Theorem 5.27:** It is decidable in polynomial-time ( $O(n^3)$ ) if a graph can knotlessly be embedded into 3-dimensional space.

## Proof (sketch):

- Robertson & Seymour proved a general result that implies the existence of a finite set of **forbidden structures** in knotlessly embeddable graphs.
- For each of these **forbidden structures** we can test whether a graph contains one of them in time  $O(n^3)$ .
- Hence, to decide if a graph is knotlessly embeddable, we only need to test for each of the finitely many **forbidden structures**, whether they occur in the graph.

This yields a cubic time decision procedure. □

**However:** We do not currently know what these structures are.

For a formal definition of knots and knotless embeddings, see [https://en.wikipedia.org/wiki/Knot\\_\(mathematics\)](https://en.wikipedia.org/wiki/Knot_(mathematics))

# Summary and Outlook

Complexity classes are based on **asymptotic resource estimates**, further generalised by considering general classes of bounds (e.g., all polynomial functions)

Ignoring constant factors is justified due to **Linear Speedup**

**P** is the most common approximation of “efficient”

**Polynomial many-one reductions** are used to show membership in **P**

## **What's next?**

- NP
- Hardness and completeness
- More examples of problems