TECHNISCHE UNIVERSITÄT DRESDEN

COMPLEXITY THEORY

Lecture 1: Introduction and Motivation

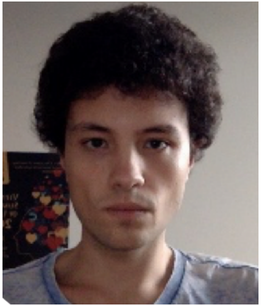
Markus Krötzsch

Knowledge-Based Systems

TU Dresden, 15th Oct 2019

Course Tutors

Markus Krötzsch
Lectures



David Carral
Exercises

Organisation

Lectures

Monday, DS 2 (9:20–10:50), APB E008
Tuesday, DS 2 (9:20–10:50), APB E005

Exercise Sessions (starting 16 October)

Wednesday, DS 3 (11:10–12:40), APB E005

Web Page

`https://iccl.inf.tu-dresden.de/web/Complexity_Theory_(WS2019/20)`

Lecture Notes

Slides of current and past lectures will be online.

Goals and Prerequisites

Goals

- Introduce basic notions of **computational complexity theory**
- Introduce **commonly known complexity classes** (P, NP, PSpace, . . .) and discuss relationships between them
- Develop **tools to classify problems** into their corresponding complexity classes
- Introduce some **advanced topics of complexity theory** (e.g., circuits, probabilistic computation, quantum computing)

(Non-)Prerequisites

- No particular prior courses needed
- Prior acquaintance with Turing Machines and basic topics in formal languages and complexity is helpful
- General mathematical and theoretical computer science skills necessary

Reading List

- **Michael Sipser: Introduction to the Theory of Computation, International Edition; 3rd Edition; Cengage Learning 2013**
- Sanjeev Arora and Boaz Barak: **Computational Complexity: A Modern Approach**; Cambridge University Press 2009
- Michael R. Garey and David S. Johnson: **Computers and Intractability**; Bell Telephone Laboratories, Inc. 1979
- Erich Grädel: **Complexity Theory**; Lecture Notes, Winter Term 2009/10
- John E. Hopcroft and Jeffrey D. Ullman: **Introduction to Automata Theory, Languages, and Computation**; Addison Wesley Publishing Company 1979
- Christos H. Papadimitriou: **Computational Complexity**; 1995 Addison-Wesley Publishing Company, Inc

Computational Problems are Everywhere

Example 1.1:

- What are the factors of 54,623?
- What is the shortest route by car from Berlin to Hamburg?
- My program now runs for two weeks. Will it ever stop?
- Is this C++ program syntactically correct?

Computational Problems are Everywhere

Example 1.1:

- What are the factors of 54,623?
- What is the shortest route by car from Berlin to Hamburg?
- My program now runs for two weeks. Will it ever stop?
- Is this C++ program syntactically correct?

Clear

Computational Problems are ubiquitous in our everyday life!

And, depending on what we want to do, those problems might be either **easily solvable** or **hardly solvable**.

Computational Problems are Everywhere

Example 1.1:

- What are the factors of 54,623?
- What is the shortest route by car from Berlin to Hamburg?
- My program now runs for two weeks. Will it ever stop?
- Is this C++ program syntactically correct?

Clear

Computational Problems are ubiquitous in our everyday life!
And, depending on what we want to do, those problems might be either **easily solvable** or **hardly solvable**.

Approach to problems:

> [T]he way is to avoid what is strong, and strike at what is weak.

(Sun Tzu: The Art of War, Chapter 6: Weak Points and Strong)

Examples

Examples

Example 1.2 (Shortest Path Problem): Given a weighted graph and two vertices s, t, find the shortest path between s and t.

Examples

Example 1.2 (Shortest Path Problem): Given a weighted graph and two vertices s, t, find the shortest path between s and t.

Easily solvable using, e.g., Dijkstra's Algorithm.

Examples

Example 1.2 (Shortest Path Problem): Given a weighted graph and two vertices s, t, find the shortest path between s and t.

Easily solvable using, e.g., Dijkstra's Algorithm.

Example 1.3 (Longest Path Problem): Given a weighted graph and two vertices s, t, find the **longest** path between s and t.

Examples

Example 1.2 (Shortest Path Problem): Given a weighted graph and two vertices s, t, find the shortest path between s and t.

Easily solvable using, e.g., Dijkstra's Algorithm.

Example 1.3 (Longest Path Problem): Given a weighted graph and two vertices s, t, find the **longest** path between s and t.

No efficient algorithm known, and believed to not exist (this problem is **NP-hard**)

Examples

Example 1.2 (Shortest Path Problem): Given a weighted graph and two vertices s, t, find the shortest path between s and t.

Easily solvable using, e.g., Dijkstra's Algorithm.

Example 1.3 (Longest Path Problem): Given a weighted graph and two vertices s, t, find the **longest** path between s and t.

No efficient algorithm known, and believed to not exist (this problem is **NP-hard**)

Observation

Difficulty of a problem is hard to assess

Measuring the Difficulty of Problems

Question

How can we measure the complexity of a problem?

Measuring the Difficulty of Problems

Question

How can we measure the complexity of a problem?

Approach

Estimate the resource requirements of the "best" algorithm that solves this problem.

Typical Resources:

- Running Time
- Memory Used

Measuring the Difficulty of Problems

Question

How can we measure the complexity of a problem?

Approach

Estimate the resource requirements of the “best” algorithm that solves this problem.

Typical Resources:

- Running Time
- Memory Used

Note

To assess the complexity of a problem, we need to consider **all possible algorithms** that solve this problem.

Problems

What actually is . . . a Problem?

(Decision) Problems are **word problems** of particular languages.

Problems

What actually is . . . a Problem?

(Decision) Problems are **word problems** of particular languages.

Example 1.4: "Problem: Is a given graph connected?" will be modelled as the word problem of the language

$$\text{GCONN} := \{ \langle G \rangle \mid G \text{ is a connected graph} \}.$$

Then for a graph G we have

$$G \text{ is connected} \iff \langle G \rangle \in \text{GCONN}.$$

Note

The notation $\langle G \rangle$ denotes a suitable encoding of the graph G over some fixed alphabet (e.g., $\{ 0, 1 \}$).

Algorithms

What actually is . . . an Algorithm?

Algorithms

What actually is … an Algorithm?

Different approaches to formalise the notion of an “algorithm”

- Turing Machines
- Lambda Calculus
- μ-Recursion
- …

What actually is . . . an Algorithm?

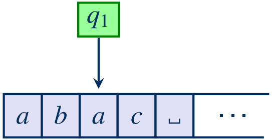
Different approaches to formalise the notion of an “algorithm”

- Turing Machines
- Lambda Calculus
- μ-Recursion
- . . .

What actually is . . . an Algorithm?

Different approaches to formalise the notion of an “algorithm”

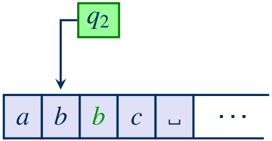
- Turing Machines
- Lambda Calculus
- μ-Recursion
- . . .



What actually is … an Algorithm?

Different approaches to formalise the notion of an "algorithm"

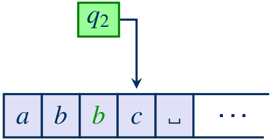
- Turing Machines
- Lambda Calculus
- μ-Recursion
- …



What actually is . . . an Algorithm?

Different approaches to formalise the notion of an “algorithm”

- Turing Machines
- Lambda Calculus
- μ-Recursion
- . . .



Avoid What is Strong

Avoid What is Strong

Suppose we are given a language $\mathcal{L}$ and a word w.

Question

Does there need to exist **any** algorithm that decides whether $w \in \mathcal{L}$?

Avoid What is Strong

Suppose we are given a language $\mathcal{L}$ and a word w.

Question

Does there need to exist **any** algorithm that decides whether $w \in \mathcal{L}$?

Answer

No. Some problems are **undecidable**.

Avoid What is Strong

Suppose we are given a language $\mathcal{L}$ and a word w.

Question

Does there need to exist **any** algorithm that decides whether $w \in \mathcal{L}$?

Answer

No. Some problems are **undecidable**.

Example 1.5:

Avoid What is Strong

Suppose we are given a language $\mathcal{L}$ and a word w.

Question

Does there need to exist **any** algorithm that decides whether $w \in \mathcal{L}$?

Answer

No. Some problems are **undecidable**.

Example 1.5:

- The Halting Problem of Turing machines

Avoid What is Strong

Suppose we are given a language $\mathcal{L}$ and a word w.

Question

Does there need to exist **any** algorithm that decides whether $w \in \mathcal{L}$?

Answer

No. Some problems are **undecidable**.

Example 1.5:

- The Halting Problem of Turing machines
- The Entscheidungsproblem (Is a first-order logical statement true?)

Avoid What is Strong

Suppose we are given a language $\mathcal{L}$ and a word w.

Question

Does there need to exist **any** algorithm that decides whether $w \in \mathcal{L}$?

Answer

No. Some problems are **undecidable**.

Example 1.5:

- The Halting Problem of Turing machines
- The Entscheidungsproblem (Is a first-order logical statement true?)
- Finding the lowest air fare between two cities (→ Reference)

Avoid What is Strong

Suppose we are given a language $\mathcal{L}$ and a word w.

Question

Does there need to exist **any** algorithm that decides whether $w \in \mathcal{L}$?

Answer

No. Some problems are **undecidable**.

Example 1.5:

- The Halting Problem of Turing machines
- The Entscheidungsproblem (Is a first-order logical statement true?)
- Finding the lowest air fare between two cities (→ Reference)
- Deciding syntactic validity of C++ programs (→ Reference)

Avoid What is Strong

Suppose we are given a language $\mathcal{L}$ and a word w.

Question

Does there need to exist **any** algorithm that decides whether $w \in \mathcal{L}$?

Answer

No. Some problems are **undecidable**.

Example 1.5:

- The Halting Problem of Turing machines
- The Entscheidungsproblem (Is a first-order logical statement true?)
- Finding the lowest air fare between two cities (→ Reference)
- Deciding syntactic validity of C++ programs (→ Reference)

Avoid: We will focus mostly on decidable problems in this course.

Time and Space

Difficulty

Measuring running time and memory requirements depends highly on the **machine**, and not so much on the **problem**.

Time and Space

Difficulty

Measuring running time and memory requirements depends highly on the **machine**, and not so much on the **problem**.

Resort

Measure time and space only **asymptotically** using **Big-$\mathcal{O}$**-Notation:

Time and Space

Difficulty

Measuring running time and memory requirements depends highly on the **machine**, and not so much on the **problem**.

Resort

Measure time and space only **asymptotically** using **Big-$\mathcal{O}$**-Notation:

$$f(n) = \mathcal{O}(g(n)) \iff f(n) \text{ “asymptotically bounded by” } g(n)$$

Difficulty

Measuring running time and memory requirements depends highly on the **machine**, and not so much on the **problem**.

Resort

Measure time and space only **asymptotically** using **Big-$\mathcal{O}$**-Notation:

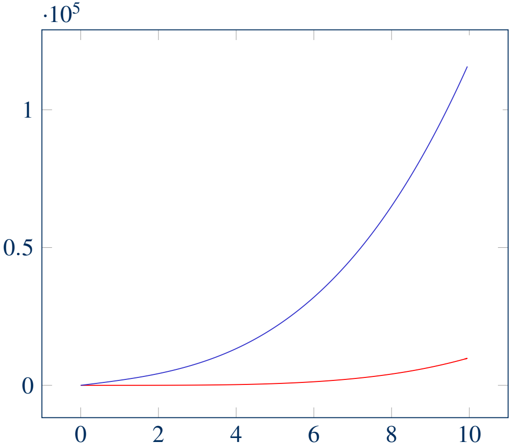
$$f(n) = \mathcal{O}(g(n)) \iff f(n) \text{ “asymptotically bounded by” } g(n)$$

More formally:

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

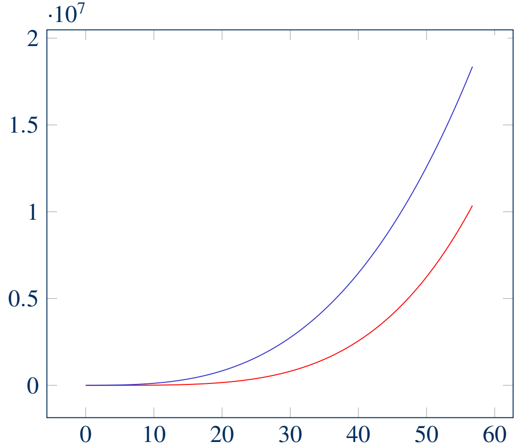
Big-$\mathcal{O}$-Notation: Example

$100n^3 + 1729n = \mathcal{O}(n^4)$:



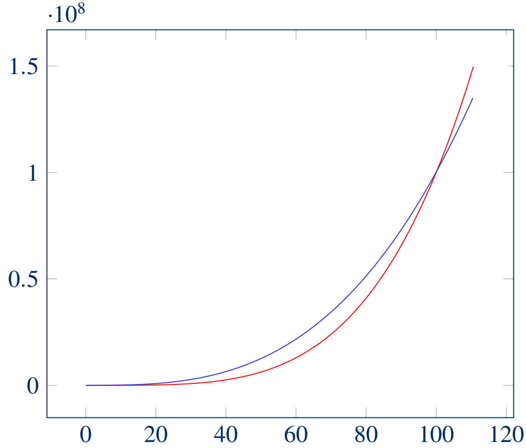
Big-$\mathcal{O}$-Notation: Example

$100n^3 + 1729n = \mathcal{O}(n^4)$:



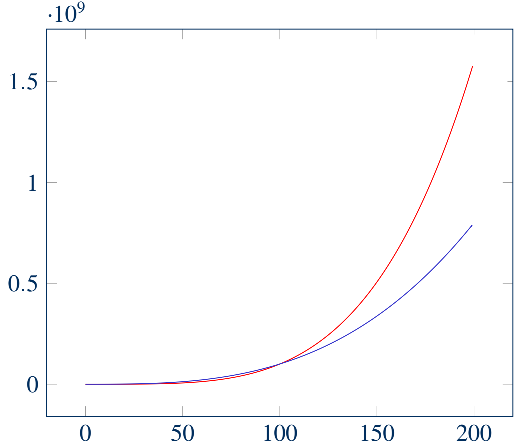
Big-$\mathcal{O}$-Notation: Example

$100n^3 + 1729n = \mathcal{O}(n^4)$:



Big-$\mathcal{O}$-Notation: Example

$100n^3 + 1729n = \mathcal{O}(n^4)$:



Complexity of Problems

Approach

The **time (space) complexity** of a problem is the asymptotic running time of a fastest (least memory consumptive) algorithm that solves the problem.

Complexity of Problems

Approach

The **time (space) complexity** of a problem is the asymptotic running time of a fastest (least memory consumptive) algorithm that solves the problem.

Problem

Still too difficult . . .

Complexity of Problems

Approach

The **time (space) complexity** of a problem is the asymptotic running time of a fastest (least memory consumptive) algorithm that solves the problem.

Problem

Still too difficult . . .

Example 1.6 (Travelling Salesman Problem): Given a weighted graph, find the shortest simple path visiting every node.

Complexity of Problems

Approach

The **time (space) complexity** of a problem is the asymptotic running time of a fastest (least memory consumptive) algorithm that solves the problem.

Problem

Still too difficult . . .

Example 1.6 (Travelling Salesman Problem): Given a weighted graph, find the shortest simple path visiting every node.

- Best known algorithm runs in time $O(n^2 2^n)$
  (Bellman-Held-Karp algorithm)

Complexity of Problems

Approach

The **time (space) complexity** of a problem is the asymptotic running time of a fastest (least memory consumptive) algorithm that solves the problem.

Problem

Still too difficult . . .

Example 1.6 (Travelling Salesman Problem): Given a weighted graph, find the shortest simple path visiting every node.

- Best known algorithm runs in time $O(n^2 2^n)$ (Bellman-Held-Karp algorithm)
- Best known lower bound is $O(n \log n)$

Complexity of Problems

Approach

The **time (space) complexity** of a problem is the asymptotic running time of a fastest (least memory consumptive) algorithm that solves the problem.

Problem

Still too difficult . . .

Example 1.6 (Travelling Salesman Problem): Given a weighted graph, find the shortest simple path visiting every node.

- Best known algorithm runs in time $O(n^2 2^n)$ (Bellman-Held-Karp algorithm)
- Best known lower bound is $O(n \log n)$
- Exact complexity of TSP unknown

Even more abstraction

Approach

Divide decision problems into the "quality" of their fastest algorithms:

Even more abstraction

Approach

Divide decision problems into the “quality” of their fastest algorithms:

- P is the class of problems **solvable in polynomial time**

Even more abstraction

Approach

Divide decision problems into the "quality" of their fastest algorithms:

- P is the class of problems **solvable in polynomial time**
- PSpace is the class of problems **solvable in polynomial space**

Even more abstraction

Approach

Divide decision problems into the “quality” of their fastest algorithms:

- P is the class of problems **solvable in polynomial time**
- PSpace is the class of problems **solvable in polynomial space**
- ExpTime is the class of problems **solvable in exponential time**

Even more abstraction

Approach

Divide decision problems into the “quality” of their fastest algorithms:

- P is the class of problems **solvable in polynomial time**
- PSpace is the class of problems **solvable in polynomial space**
- ExpTime is the class of problems **solvable in exponential time**
- L is the class of problems **solvable in logarithmic space** (apart from the input)

Even more abstraction

Approach

Divide decision problems into the “quality” of their fastest algorithms:

- P is the class of problems **solvable in polynomial time**
- PSpace is the class of problems **solvable in polynomial space**
- ExpTime is the class of problems **solvable in exponential time**
- L is the class of problems **solvable in logarithmic space** (apart from the input)
- NP is the class of problems **verifiable in polynomial time**

Even more abstraction

Approach

Divide decision problems into the “quality” of their fastest algorithms:

- P is the class of problems **solvable in polynomial time**
- PSpace is the class of problems **solvable in polynomial space**
- ExpTime is the class of problems **solvable in exponential time**
- L is the class of problems **solvable in logarithmic space** (apart from the input)
- NP is the class of problems **verifiable in polynomial time**
- NL is the class of problems **verifiable in logarithmic space**

Even more abstraction

Approach

Divide decision problems into the "quality" of their fastest algorithms:

- P is the class of problems **solvable in polynomial time**
- PSpace is the class of problems **solvable in polynomial space**
- ExpTime is the class of problems **solvable in exponential time**
- L is the class of problems **solvable in logarithmic space** (apart from the input)
- NP is the class of problems **verifiable in polynomial time**
- NL is the class of problems **verifiable in logarithmic space**

And many more!

Even more abstraction

Approach

Divide decision problems into the "quality" of their fastest algorithms:

- P is the class of problems **solvable in polynomial time**
- PSpace is the class of problems **solvable in polynomial space**
- ExpTime is the class of problems **solvable in exponential time**
- L is the class of problems **solvable in logarithmic space** (apart from the input)
- NP is the class of problems **verifiable in polynomial time**
- NL is the class of problems **verifiable in logarithmic space**

And many more!
$\oplus$P, #P, AC, AC^0, ACC0, AM, AP, APSpace, BPL, BPP, BQP, coNP, E, Exp, FP, IP, MA, MIP, NC, NExpTime, P/poly, PH, PP, RL, RP, Σ_i^{p}, TISP$(T(n), S(n))$, ZPP, ...

Strike at What is Weak

Approach (cf. Cobham–Edmonds Thesis)

The problems in P are "tractable" or "efficiently solvable"
(and those outside are not)

Strike at What is Weak

Approach (cf. Cobham–Edmonds Thesis)

The problems in P are “tractable” or “efficiently solvable”
(and those outside are not)

Example 1.7: The following problems are in P:

- Shortest Path Problem
- Satisfiability of Horn-Formulas
- Linear Programming
- Primality

Strike at What is Weak

Approach (cf. Cobham–Edmonds Thesis)

The problems in P are "tractable" or "efficiently solvable"
(and those outside are not)

Example 1.7: The following problems are in P:

- Shortest Path Problem
- Satisfiability of Horn-Formulas
- Linear Programming
- Primality

Note

The Cobham-Edmonds-Thesis is only a **rule of thumb**: there are (practically) tractable problems outside of P, and (practically) intractable problems in P.

Friend or Foe?

Caveat

It is not known how big P is.

In particular, it is unknown whether P $\neq$ NP or not.

Friend or Foe?

Caveat

It is not known how big P is.
In particular, it is unknown whether P $\neq$ NP or not.

Approach

Try to find out which problems in a class are at least as hard as others.

Friend or Foe?

Caveat

It is not known how big P is.
In particular, it is unknown whether P $\neq$ NP or not.

Approach

Try to find out which problems in a class are at least as hard as others.
Complete problems are then the hardest problems of a class.

Friend or Foe?

Caveat

It is not known how big P is.
In particular, it is unknown whether $\mathsf{P} \neq \mathsf{NP}$ or not.

Approach

Try to find out which problems in a class are at least as hard as others.
Complete problems are then the hardest problems of a class.

Example 1.8: Satisfiability of propositional formulas is **NP-complete**: if we can efficiently decide whether a propositional formula is satisfiable, we can solve **any** problem in NP efficiently.

Friend or Foe?

Caveat

It is not known how big P is.
In particular, it is unknown whether P $\neq$ NP or not.

Approach

Try to find out which problems in a class are at least as hard as others.
Complete problems are then the hardest problems of a class.

Example 1.8: Satisfiability of propositional formulas is **NP-complete**: if we can efficiently decide whether a propositional formula is satisfiable, we can solve **any** problem in NP efficiently.

But: we still do not know whether we can or cannot solve satisfiability efficiently. We only know it will be difficult to find out . . .

Learning Goals

- Get an overview over the foundations of Complexity Theory
- Gain insights into advanced techniques and results in Complexity Theory
- Understand what it means to “compute” something, and what the strengths and limits of different computing approaches are
- Get a feeling of how hard certain problems are, and where this hardness comes from
- Appreciate how very little we actually know about the computational complexity of many problems

Lecture Outline (1)

- **Turing Machines** (Revision)
  Definition of Turing Machines; Variants; Computational Equivalence; Decidability and Recognizability; Enumeration; Oracles
- **Undecidability**
  Examples of Undecidable Problems; Mapping Reductions; Rice's Theorem; Recursion Theorem
- **Time Complexity**
  Measuring Time Complexity; Many-One Reductions; Cook-Levin Theorem; Time Complexity Classes (P, NP, ExpTime); NP-completeness; pseudo-NP-complete problems
- **Space Complexity**
  Space Complexity Classes (PSpace, L, NL); Savitch's Theorem; PSpace-completeness; NL-completeness; NL = coNL

Lecture Outline (2)

- **Diagonalisation**
  Hierarchy Theorems (det. Time, non-det. Time, Space); Gap Theorem; Ladner's Theorem; Relativisation; Baker-Gill-Solovay Theorem
- **Alternation**
  Alternating Turing Machines; APTime = PSpace; APSpace = ExpTime; Polynomial Hierarchy
- **Circuit Complexity**
  Boolean Circuits; Alternative Proof of Cook-Levin Theorem; Parallel Computation (NC); P-completeness; P$/$poly; (Karp-Lipton Theorem, Meyer's Theorem)
- **Probabilistic Computation**
  Randomised Complexity Classes (RP, PP, BPP, ZPP); Sipser-Gács-Lautemann Theorem
- **Quantum Computing**
  Quantum mechanics for computer scientists, entanglement, quantum circuits, BQP

Avoid what is Strong, and Strike at what is Weak

Sometimes the best way to solve a problem is to avoid it . . .

Avoid what is Strong, and Strike at what is Weak

Sometimes the best way to solve a problem is to avoid it . . .