Theoretical Computer Science

Complexity Theory

Technische Hochschule Rosenheim
Sommer 2022
Prof. Dr. Jochen Schmidt

Overview

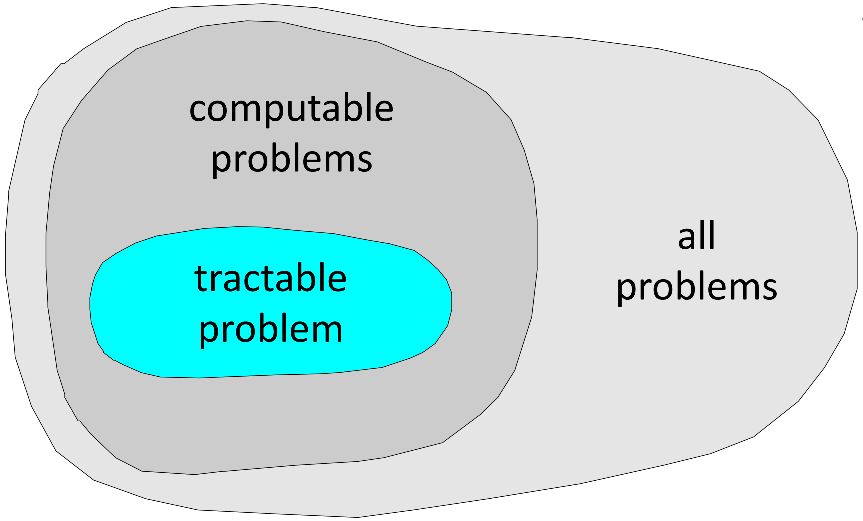
- Time and space complexity
- Order of complexity, O-Notation
- Optimization using the example of divide and conquer
- Complexity Classes P, NP
- NP completeness & NP hard problems
- Other problem classes

Introduction

- Previous chapter: computability
  - is a problem in principle solvable with computers – does an algorithm exist?
- Now: How much effort is required for solving a computable problem, in particular
  - Time complexity (how much computation time is required dependent on amount of input data?)
  - Space complexity (how much memory is required dependent on amount of input data?)
- In the following: Mainly time complexity
  - space complexity is considered using the same methods & notations
  - but is often less important in practice

Time Complexity

- Only a part of the computable problems is tractable
- The others take too long or require too much memory for practical purposes



We can consider the time complexity of

- a specific algorithm: Number of steps required to solve a problem.
- a problem: Time complexity required by an optimal algorithm for solving the problem.

Time Complexity – Variants

- Worst-case complexity
  - how "long" does the algorithm take maximum (for the "worst-case" structure of input data)
  - often: complexity = worst-case complexity
- Best-case complexity
  - how "long" does the algorithm take at least (with optimal structure of input data)
- Average-case complexity
  - expected complexity with a typical structure of data ("average runtime")

Example: Linked list containing 20 (or in general: n) family names, search for a name

Name is in last element $\longrightarrow$ 20 steps or n steps

Name is in first element $\longrightarrow$ 1 step

Name is in middle element $\longrightarrow$ 10 steps or $\frac{n}{2}$ steps

Order of Complexity

Time Complexity – Objectives

- Dependent on the size of the input data ⟶ Parameter n
  - How does the algorithm behave when the amount of input data increases?
  - We want a function. Not a measured time.

- Omit “unimportant" constants
  - Constant factors such as: computer hardware used, programming language used and its compiler, or clock frequency of the CPU
  - Complexity should only depend on the algorithm, not on the hardware used

- We will look at an upper bound („asymptotic time complexity“)

O-Notation – Definition

- $O\big(f(n)\big) = \{g: \mathbb{N} \to \mathbb{N} \mid \exists m > 0, c > 0 \text{ where } \forall n \geq m: |g(n)| \leq c \cdot |f(n)|\}$
- i.e., $O\big(f(n)\big)$ is the set of all functions $g(n)$,
  - for which there exist the two constants m, c,
  - such that for all $n \geq m$ it holds that $|g(n)| \leq c \cdot |f(n)|$
- in other words: $g(n)$ grows at most as fast as $f(n)$
- and this applies asymptotically, i.e., from a certain point on, for $n \longrightarrow \infty$

- Usual notation: $g(n) = O(f(n))$
  - e.g., $g(n) = O(n^2)$
  - technically not correct: „$\in$“ should be used (it is a set)
  - Problem: The operator = is not symmetrical here:
    $O(n) = O(n^2)$ is true, but not $O(n^2) = O(n)$

O-Notation – Examples

- $f(n) = 50n + 3 = O(n)$
  - $c = 51,\ m = 3$
- $f(n) = 2n^2 - 50n + 3 = O(n^2)$
  - $|\ 2n^2 - 50n + 3\ | \leq 2n^2 + |50n| + 3 \leq$ $2n^2 + 50n^2 + 3n^2 = 55n^2 = |55n^2|$
  - therefore $|\ 2n^2 - 50n + 3\ | \leq 55\ |n^2|$
  - and thus: $c = 55,\ m = 1$
- In general:
  - only the fastest growing term is relevant
  - all slower growing terms and constant factors are omitted

- $f(n) = 3 \ln n = O(\ln n)$
- $f(n) = \ln n^c = O(\ln n)$
  - $\ln n^c = c \ln n$ $\longrightarrow$ constant factor
- $f(n) = 3 \log_2 n = O(\ln n)$
  - $\log_2 n = \ln n / \ln 2$ $\longrightarrow$ constant factor
- In general:
  - Base of a logarithm is irrelevant
  - Constant exponents under the logarithm are irrelevant

O-Notation – Examples

- $f(n) = \log n - 3n + 2n^3 + 2^n = O(2^n)$
- $f(n) = \log n - 3n + 2n^3 + 10^n = O(10^n)$
- $f(n) = \log n - 3n + 2n^3 + 2^n + 10^n = O(10^n)$
- In general:
  Changing the base of an exponential function is relevant

- $f(n) = 50n + 3 = O(2^n)$
- $f(n) = 2n^2 - 50n + 3 = O(2^n)$
- $f(n) = \ln n - 3n + 2n^3 = O(2^n)$
- $f(n) = 3 \ln n = O(2^n)$
- In general:
  - above statements are correct, but not very helpful
  - we are looking for a tight upper bound

Landau Symbols

- introduced by Paul Bachmann 1894
- named after Edmund Landau (1877 – 1938)
- Here: Two other symbols in addition to O-notation (Ω, Θ)

$g = O(f)$	g grows at most as fast as f (upper bound)	$\lvert g(n)\rvert \leq c \cdot \lvert f(n)\rvert$
$g = \Omega(f)$	g grows at least as fast as f (lower bound)	$\lvert g(n)\rvert \geq c \cdot \lvert f(n)\rvert$
$g = \Theta(f)$	g grows just as fast as f	$c_0 \cdot \lvert g(n)\rvert \leq \lvert f(n)\rvert \leq c_1 \cdot \lvert g(n)\rvert$

Typical Orders of Complexity

Name	Complexity	Rating	Examples	Typical Algorithm Structure
Constant complexity	$O(1)$	optimal, rare	Hashing	Most statements are executed only once or a few times.
Logarithmic complexity	$O(\log n)$	very good	Binary search in sorted list	Solve a problem by converting it to a smaller one, while reducing the runtime by a constant proportion.
Linear complexity	$O(n)$	good	Linear search in unsorted list	Optimal case for an algorithm that has to process n input data – each element must be touched exactly once (or constantly often).
Log-linear or quasilinear complexity	$O(n \log n)$	still good	Good sorting sort methods, e.g., Mergesort, Quicksort (on average); FFT	Solve a problem by splitting it into smaller problems that are solved independently and then combined.
Quadratic complexity	$O(n^2)$	poor	Poor sorting methods, e.g., Bubblesort, Quicksort (worst case)	Typical for problems where all n elements need to be processed in pairs (2 nested for loops). Can only be used for relatively small problems.
Cubic complexity	$O(n^3)$	poor	Matrix-Multiplication	3 nested for loops. Can only be used for small problems.
Exponential complexity	$O(a^n)$	disastrous	Travelling-Salesman (cleverly implemented)	Typical for brute-force solutions, e.g., trying out all possible variants. Only few algorithms of this complexity can be used in practice.
Factorial complexity	$O(n!)$	even worse…	Travelling-Salesman (brute-force)	

Note: a^n grows faster than **any** polynomial n^k for any $a > 1$

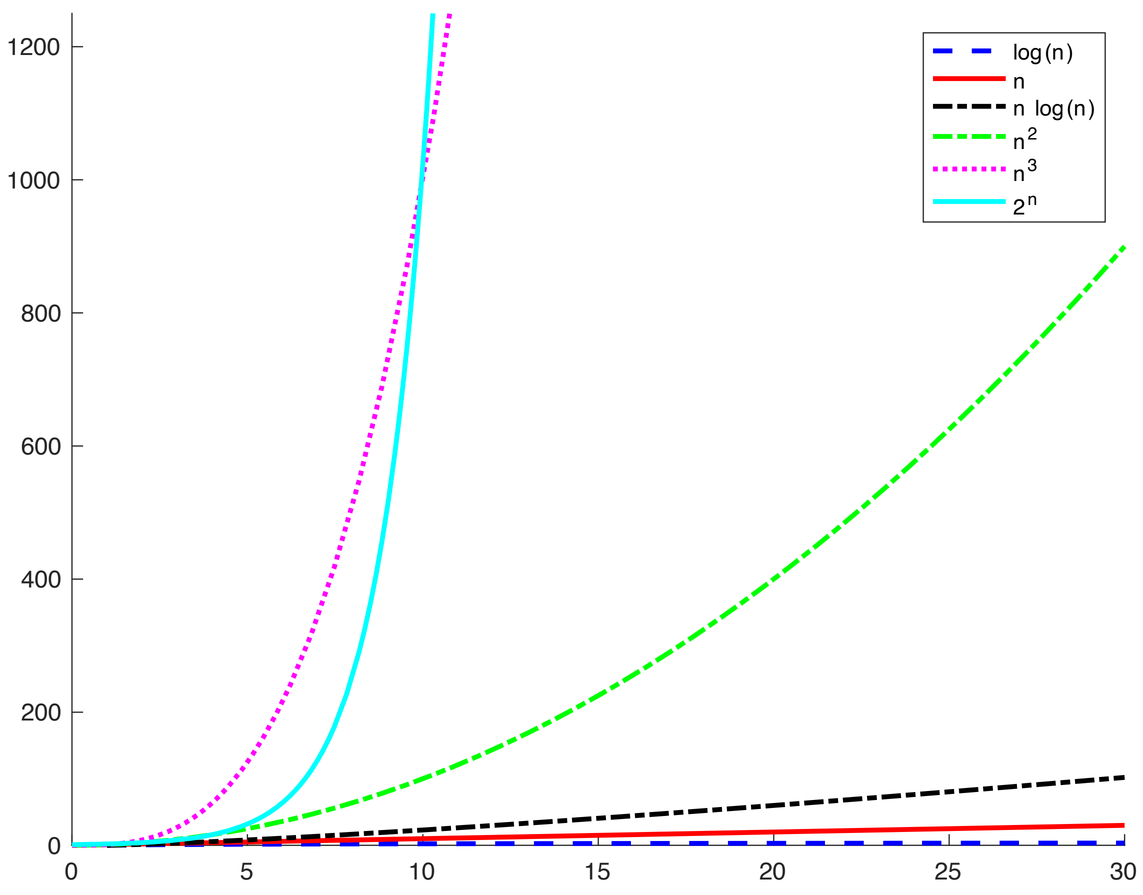
O-Notation – Examples

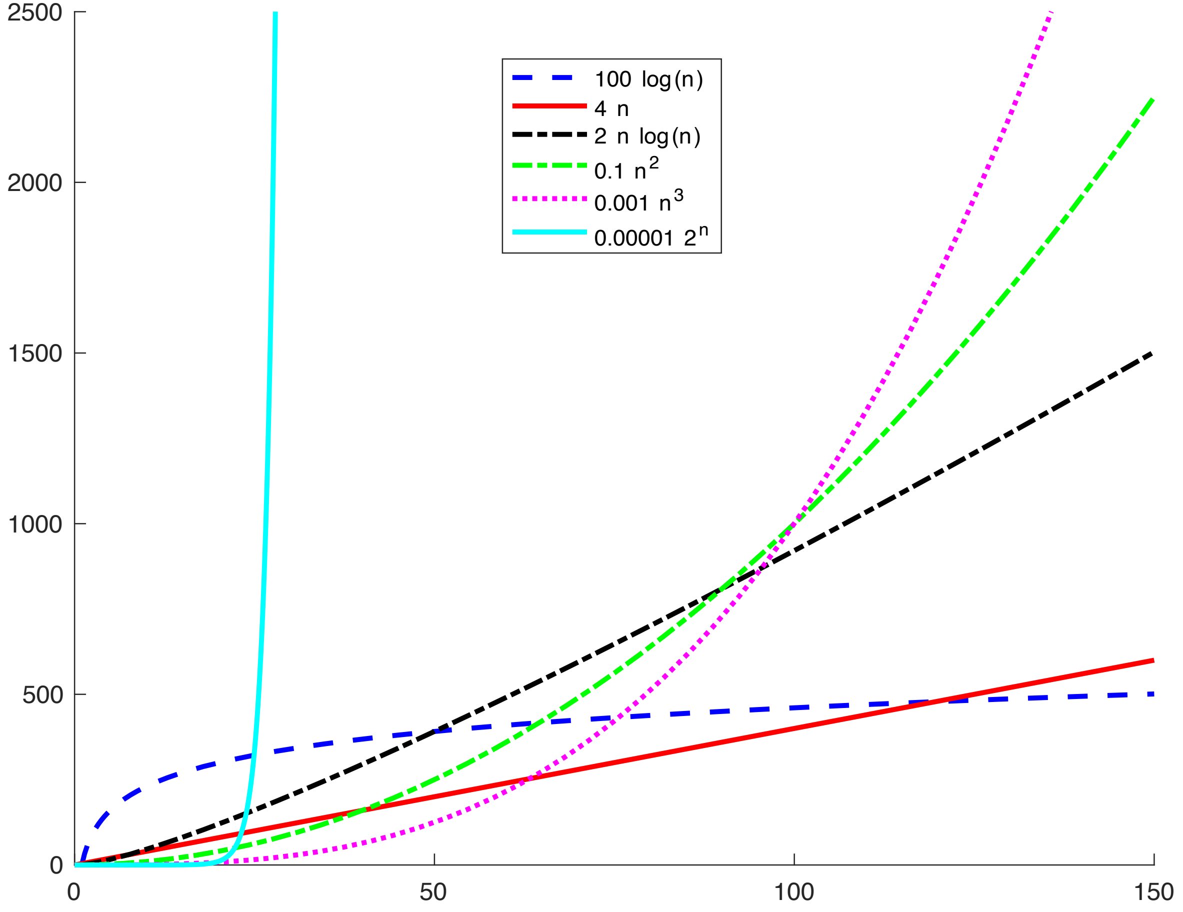
n	O(n)	O(n^2)	O(2^n)
1	1 μsec	1 μsec	2 μsec
10	10 μsec	100 μsec	~ 1 msec
100	100 μsec	10 msec	~ $4 \cdot 10^{16}$ years
1000	1 msec	1 sec	~ $8 \cdot 10^{288}$ years

Caution: the O-notation applies only asymptotically for $n \to \infty$

n	O($100 \cdot n$) = O(n)	O($0.1 \cdot n^2$) = O(n^2)	O($0.0001 \cdot 2^n$) = O(2^n)
1	100 μsec	0.1 μsec	0.0002 μsec
10	1 msec	10 μsec	~ 0.1 μsec
100	10 msec	1 msec	~ $4 \cdot 10^{12}$ years
1000	100 msec	100 msec	~ $8 \cdot 10^{284}$ years

Function Growth – Examples





Complexity in O-Notation from Code

Basic instructions (no function calls):

```
a = 15;
```

O(1)

```
x = x * a;
```

O(1)

If we combine these two instructions, we get
O(1) + O(1) = O(2) = O(1)

No matter how many basic instructions, it's O(1)

Nested loops:
(n = amount of data, B = block of constant time complexity O(1))

```
for(int i = 0; i < n; i++) {
  B;
}
```

$O(n)$

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n; j++) {
    B;
  }
}
```

$O(n^2)$

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < i; j++) {
    B;
  }
}
```

$O(n^2)$

```
for(int i = 0; i < n; i++) {
  for(int j = 0; j < 100; j++) {
    B;
  }
}
```

$O(n)$

Complexity in O-Notation from Code

Loop, changing loop counter using * or / instead of + or –:

```
i = 1;
while(i < n) {
  B;
  i = i * 2;
}
```

$O(\log_2 n) = O(\log n)$

Recursion:

```
int factorial(int n) {
  if (n == 0)
    return 1;
  else
    return n * factorial(n-1);
}
```

$O(n)$

Recursion is often hard to analyze

```
int doSomething(int a, int b) {
// Pre-condition: a < b
  if (a == b)
    return 0;
  else
    return (doSomething(a+1, b) - doSomething(a, b-1));
}
```

$O(2^n)$ (this recursion results in a binary call-tree)

Technische Hochschule Rosenheim

- Typical variants of recursion are considered in the following
  - see also: Divide-and-Conquer formula later in this set of slides
- Formulas for calculating complexity C_n are given independently of a specific algorithm
- It holds: n = amount of input data, C_n = number of steps required in total, $C_0 = 0$

- Variant 1
  - Loop over input data in each step
  - One element is removed before recursive call

$$C_n = C_{n-1} + n$$

unfold recursion:

$$C_n = 0 + 1 + 2 + \ldots + (n-3) + (n-2) + (n-1) + n = \tfrac{1}{2}\, n(n+1) = O(n^2 / 2) = O(n^2)$$

Recursion

- Variant 2
  - Loop over input data in each step
  - Input data are halved before recursion

$$C_n = C_{n/2} + n$$
$$C_n = \ldots + n/8 + n/4 + n/2 + n =$$
$$= (\ldots + 1/8 + 1/4 + 1/2 + 1)\ n$$
$$= O(2n) = O(n)$$

- Variant 3
  - Effort within a step is constant (= independent of input data size)
  - Input data are halved before recursion

$$C_n = C_{n/2} + 1$$
$$C_n = \underbrace{1 + 1 + \ldots + 1}_{\log_2 n \text{ times}} = O(\log_2 n) = O(\log n)$$

- Variant 4
  - Loop over input data in each step
  - Input data are split into two halves before recursion

$$C_n = 2C_{n/2} + n$$
$$C_n = O(n \log n)$$

- Variant 5
  - Effort within a step is constant (= independent of input data size)
  - Input data are split into two halves before recursion

$$C_n = 2C_{n/2} + 1$$
$$C_n = O(2n) = O(n)$$

Calculation Rules for O-Notation

Let c and a_i be constants.

- c = O(1)
- $c \cdot f(n)$ = O($f(n)$)
- O($f(n)$) + O($f(n)$) = O($f(n)$)
- O(O($f(n)$)) = O($f(n)$)
- $g(n) = a_k \cdot n^k + a_{k\text{-}1} \cdot n^{k\text{-}1} + \ldots + a_0$ = O(n^k)
- O($f(n)$) · O($g(n)$) = O($f(n) \cdot g(n)$)
- O($f(n)$) + O($g(n)$) = O($\max\{f(n), g(n)\}$)

Application to Analysis of Algorithms: How to obtain the total complexity from parts?

Basic instructions are O(1)

Iteration n-times in a loop with body O(alg): O(n · alg)

Sequences alg1; alg2; alg3;
O(alg1) + O(alg2) + O(alg3) = O(max{alg1, alg2, alg3})

IF THEN alg 1 ELSE alg2
O(max{alg1, alg2})

Algorithm Optimization Using the Example of Divide and Conquer

Objective

- Find a better algorithm to solve a given problem
  - better = lower time (or sometimes space) complexity
- Optimization is problem dependent,
- and also, what we count as a relevant operation for time complexity
  - Mathematical algorithms (as the following example): Number of arithmetic operations
    - We'll count multiplications and additions
    - In the past, typically only multiplications were counted (as they used to be many times slower)
  - For searching and sorting: Count number of comparisons required

Example: Evaluation of Polynomials

- How can we compute the value of a polynomial function f(x) at b, i.e., f(b)?
  $f(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$

- Complexity "naïve" method for calculating f(b):
  - Calculation of powers $x^2, \ldots, x^n$:
    - $2 + 3 + 4 + \ldots + n = n\,(n + 1) / 2 - 1$ multiplications: $O(n^2/2)$
  - n multiplications by coefficients a_i
  - n additions
  - result: $n\,(n + 1) / 2 - 1 + 2n = O(n^2/2 + 2n) = O(n^2)$

- Re-using powers that have already been calculated
  - in each step only one additional multiplication, total: $n - 1$
  - results in $2n - 1$ multiplications and n additions
  - total: $3n - 1 = O(3n) = O(n)$

Example: Evaluation of Polynomials

- Horner's method
  - factoring out: $f(x) = a_0 + x\,(a_1 + x\,(a_2 + x\,(a_3 + \ldots + x\,(a_{n-1} + a_n x)\ldots)$

- Complexity
  - n multiplications and n additions
  - (due to O-notation it doesn't matter whether it is actually n – 1 or n – 2 or …)
  - total: $O(2n) = O(n)$

- Note: The Fast Fourier Transform (FFT) is better still:
  It can be used to evaluate a polynomial at n positions in parallel with $O(n \log n)$

Divide and Conquer (*Teile und Herrsche*)

- Divide and Conquer:
  - A very important algorithm development paradigm
  - Break down a problem into non-overlapping sub-problems
  - Combine the individual solutions to a complete solution
  - Do this recursively

- Often:
  - Divide data into two parts
  - process these separately

- Examples
  - Quicksort, Mergesort
  - Karatsuba's method for multiplying long integers
  - Fast Fourier transform (FFT)

Divide and Conquer – How to Obtain the Complexity

Technische Hochschule **Rosenheim**

- Effort to break down a problem of size n into a sub-problems of size n/b:

$$C(n) = a\ C(n/b) + \Theta(n^k) \qquad \text{for } a \geq 1,\ \ b, n > 1$$
$$C(1) = 1$$

- $\Theta(n^k)$: Effort for splitting and combining data in each step

- $C(n)$ can be estimated as follows:

$$C(n) = \begin{cases} \Theta(n^k) & \text{if } a < b^k \\ \Theta(n^k \log n) & \text{if } a = b^k \\ \Theta(n^{\log_b a}) & \text{if } a > b^k \end{cases}$$

Divide and Conquer – Complexity Examples

$C(n) = a\ C(n/b) + \Theta(n^k)$

$$C(n) = \begin{cases} \Theta(n^k) & \text{if } a < b^k \\ \Theta(n^k \log n) & \text{if } a = b^k \\ \Theta(n^{\log_b a}) & \text{if } a > b^k \end{cases}$$

- $C(n) = 2\ C(n/2) + O(n) \quad \rightarrow O(n \log n)$
- $C(n) = 2\ C(n/2) + O(n^2) \quad \rightarrow O(n^2)$

- $C(n) = 8\ C(n/3) + O(n^2) \quad \rightarrow O(n^2)$
- $C(n) = 9\ C(n/3) + O(n^2) \quad \rightarrow O(n^2 \log n)$
- $C(n) = 10\ C(n/3) + O(n^2) \quad \rightarrow O\left(n^{\log_3 10}\right) = O(n^{2.09})$

Example: Integer Multiplication

- Multiplication of two integers with n digits each as taught in school
- Example:

```
 1 2 3 4 5   ·   6 7 8 9 0
-----------------------------
       7 4 0 7 0
         8 6 4 1 5
           9 8 7 6 0
           1 1 1 1 0 5
 +               0 0 0 0 0
-----------------------------
       8 3 8 1 0 2 0 5 0
```

Complexity: $O(n^2)$ – corresponds to the size of the table

Can we do better?

Karatsuba's Integer Multiplication

- by Karatsuba and Ofman (1962)
- Idea: Split the n-digit integers A and B into two parts:
  - In the middle, at position n/2
- $A = a_1 10^{n/2} + a_2$ and $B = b_1 10^{n/2} + b_2$
- Product:

$$AB = (a_1 10^{n/2} + a_2)(b_1 10^{n/2} + b_2)$$
$$= a_1 b_1 10^n + (a_1 b_2 + a_2 b_1)\, 10^{n/2} + a_2 b_2$$

  - 4 n/2-digit multiplications
  - Combination of results:
    - Shift by n/2 and n digits, respectively
    - Addition

A:	a_1	a_2
B:	b_1	b_2

Complexity: $C(n) = 4\ C(n/2) + \mathrm{O}(n)$

$C(n) = a\ C(n/b) + \Theta(n^k)$

$$C(n) = \begin{cases} \Theta(n^k) & \text{if } a < b^k \\ \Theta(n^k \log n) & \text{if } a = b^k \\ \Theta(n^{\log_b a}) & \text{if } a > b^k \end{cases} \quad 4 > 2^1$$

$\log_2 4 = 2 \longrightarrow \mathrm{O}(n^2)$ – same as before…

Karatsuba's Integer Multiplication

Further re-formulization:

$$
\begin{aligned}
AB &= (a_1 10^{n/2} + a_2)\,(b_1 10^{n/2} + b_2)\\
&= a_1 b_1 10^n + (a_1 b_2 + a_2 b_1)\, 10^{n/2} + a_2 b_2\\
&= a_1 b_1 10^n + ((a_1 + a_2)(b_1 + b_2) - a_1 b_1 - a_2 b_2) 10^{n/2} + a_2 b_2
\end{aligned}
$$

- 3 n/2-digits multiplications (instead of 4)
- Combination of results:
  - Shift by n/2 and n digits, respectively
  - Addition

Complexity: $C(n) = 3\ C(n/2) + \mathrm{O}(n)$

$$C(n) = a\ C(n/b) + \Theta(n^k)$$

$$C(n) = \begin{cases} \Theta(n^k) & \text{if } a < b^k \\ \Theta(n^k \log n) & \text{if } a = b^k \\ \Theta(n^{\log_b a}) & \text{if } a > b^k \quad 3 > 2^1 \end{cases}$$

$\log_2 3 = 1.585 \longrightarrow \mathrm{O}(n^{1.585})$

Karatsuba's Integer Multiplication – Notes

- Of course, this applies to any number system
  - and is typically implemented in base 2
- You can still do better:
  - does not have a big impact in practice, however
  - Schönhage-Strassen (1971): $O(n \log n \log \log n)$

  - Fürer (2007): $O(n \operatorname{ld} n\, 2^{O(\operatorname{ld}^* n)})$
    - where $\operatorname{ld}^* n$ = the smallest i, for which $\operatorname{ld} \operatorname{ld} \ldots \operatorname{ld} n \leq 1$, where i = #times ld (base 2 log) has been used
    - Examples:
      - $\operatorname{ld}^* 2 = 1$, $\operatorname{ld}^* 4 = 2$, $\operatorname{ld}^* 16 = 3$, $\operatorname{ld}^* 65536 = 4$
    - Publication: https://wwwmath.uni-muenster.de/u/cl/WS2007-8/mult.pdf

  - Covanov and Thomé (2016): $O(n \operatorname{ld} n\, 2^{2\operatorname{ld}^* n})$
    - Publication: https://arxiv.org/abs/1502.02800

  - Harvey and van der Hoeven (2018):
    $O(n \operatorname{ld} n\, 2^{2\operatorname{ld}^* n})$ is a lower bound for complexity
    - Publication: https://arxiv.org/abs/1802.07932

Complexity Classes P – NP

Introduction

- The existence of an algorithm is no guarantee that the problem can be solved in practice
  - computation time or memory (space complexity) may be too high to be useful

- Questions:
  - Which complexity orders are still acceptable?
  - Can we define a class of tractable problems?

Introduction

Technische Hochschule Rosenheim

Problem size that can be handled in 1 hour

Complexity	Problem Size Today	Using a 100x Faster Computer	Using a 1000x Faster Computer
n	N_1	$100\ N_1$	$1000\ N_1$
n^2	N_2	$10\ N_2$	$32\ N_2$
n^3	N_3	$4{,}6\ N_3$	$10\ N_3$
n^5	N_4	$2{,}5\ N_4$	$4\ N_4$
2^n	N_5	$N_5 + 6{,}6$	$N_5 + 10$
3^n	N_6	$N_6 + 4{,}2$	$N_6 + 6{,}3$

Observation: With exponential complexity, a faster computer is practically useless!

Complexity Class P

- A decision problem is called efficiently solvable (or tractable) if there is an algorithm with time complexity O(p(n))
  - p(n) is a polynomial of any degree
  - i.e., algorithms with polynomial runtime

- Class P contains all decision problems that can be solved by a deterministic Turing Machine in polynomial time, i.e., all tractable problems

Class NP

Technische Hochschule Rosenheim

- Class NP contains all decision problems that can be solved by a nondeterministic Turing Machine in polynomial time
- NP stands for Nondeterministic Polynomial time

- obviously: P $\subseteq$ NP
  - any deterministic TM is also a nondeterministic TM that has no choice in state transitions
  - however, a nondeterministic TM can, in polynomial time
    - “Guess” an exponential number of solutions
    - and check them in parallel

- NP contains all efficiently verifiable decision problems
  - the nondeterministic TM “guesses” the solution in polynomial time
  - which can then be checked for correctness in polynomial time by a deterministic TM

Example: Prime Factorization

- Given: A natural number n
- Sought: Decomposition into prime factors
  - or, weaker: Integer Factorization – decomposition into integer factors
- Factorization is time-consuming: What are the prime factors of 8633?
- Verification of a solution is easy:
  - Factors: $89 \cdot 97 = 8633$

- Notes:
  - The problem above is not a decision problem, but can easily be formulated as one: Does n have a prime factor smaller than some integer k?
  - Whether prime factorization actually is difficult is an open problem … it's probably not in P, but also not as complicated as some other problems (the NP-complete ones)

Example: Boolean Satisfiability Problem (SAT)

- *Erfüllbarkeitsproblem der Aussagenlogik*
- Given: Propositional logic formula ((*Aussagen-)logischer Ausdruck*): AND $\wedge$, OR $\vee$, NOT $\neg$
- Sought: Are there variable values for which the expression is "true"?

- Searching is time-consuming: $(\neg x_1 \vee x_2) \wedge x_3 \wedge (x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3)$
- Checking is easy: $x_1 = 0, x_2 = 1, x_3 = 1$
- Note: this problem is proven to be difficult…

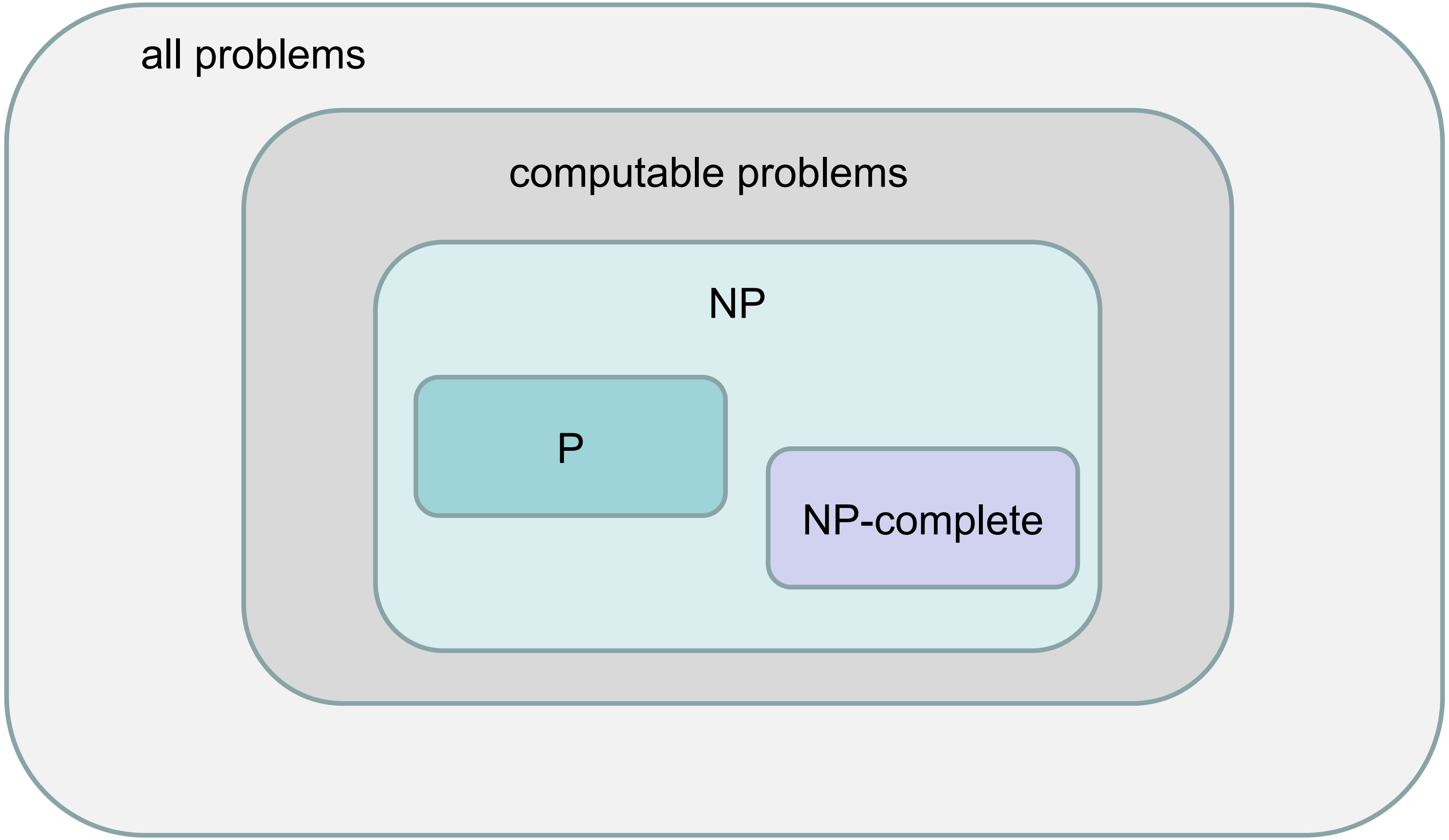
P vs NP Problem: P = NP?

- Arguably **the** most important question of theoretical computer science: Is P = NP?
  - are the two problem classes perhaps not different at all?
  - this problem has been open since the 1970s and has not yet been solved
  - it was added to the list of Millennium Problems in 2000
    - contains 7 unsolved problems of mathematics (6 of them still open)
    - a prize money of 1 million US dollars is offered for the proof https://www.claymath.org/millennium-problems/p-vs-np-problem

- Significance
  - there are a lot of problems
    - of which you can easily show that they are in NP
    - for which, however, no polynomial algorithm is known so far
  - it could be that we just have not found one yet (P = NP)
    - This would mean: All efficiently verifiable decision problems are efficiently solvable
  - or hat none exists (P ≠ NP)

- General belief: P ≠ NP

Overview



Assumption: P ≠ NP

NP-hard & NP-complete

- A problem X is called NP-hard (*NP-schwer*) if it is at least as difficult as any problem in NP
  - i.e., for all problems $L \in NP$: $L \leq_p X$ (polynomial-time reduction)

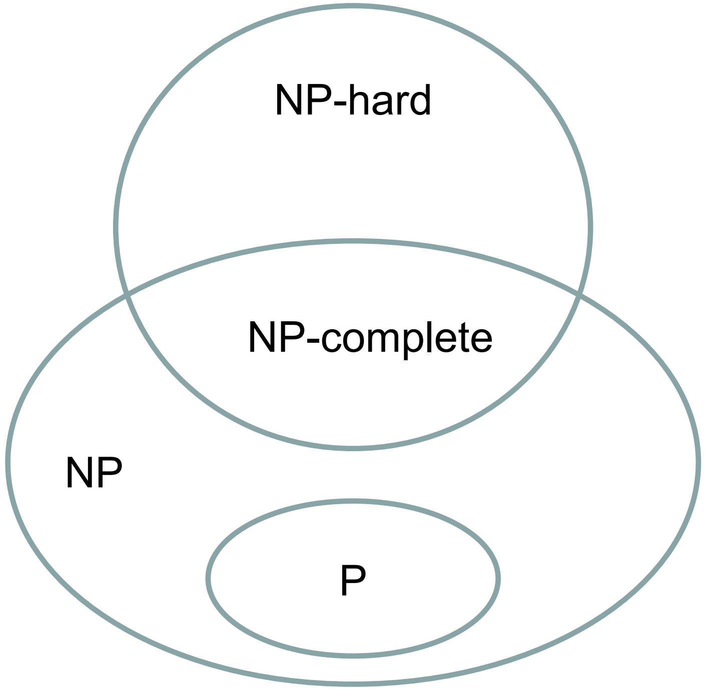
- A problem X is called NP-complete (*NP-vollständig*) if it is NP-hard and is in NP

- Polynomial-time Reduction (*polynomielle Reduktion*)
  - A decision problem A is called polynomial-time reducible to B if there is an algorithm f with polynomial complexity that transforms an input x of A into an input f(x) of B, such that both result in the same output:
    We can solve A by transforming its input it in polynomial time to B, and then solve B instead.

  - Notation: $A \leq_p B$
  - In particular, this means:
    - if $A \leq_p B$ and B is efficiently solvable ($B \in P$), then A is also efficiently solvable ($A \in P$)
    - if $A \leq_p B$ and B is efficiently verifiable ($B \in NP$), then A is also efficiently verifiable ($A \in NP$)

- Note:
  - no exact equivalence transformation of problems A and B is required
  - but: both give the same solution, i.e., a “yes” or “no” answer (A & B are decision problems)

NP-hard & NP-complete



Assumption: P ≠ NP

- NP-complete = the most difficult problems of class NP
- If there is even a single NP-complete problem in P, then P = NP
  - as all problems in NP can then be reduced to it polynomially
  - proof of a problem as being NP-complete thus is practically synonymous with the fact that there are (likely) no efficient algorithms for this problem
- if we have a first NP-complete problem C, we can show the NP-completeness of other problems X by polynomial-time reduction of X to C: $C \leq_p X$

SAT

- Do NP-complete problems exist at all?
- Yes: Boolean Satisfiability Problem (SAT)
  - the first problem proven to be NP-complete
  - Proof 1971 by S. Cook
    - „The Complexity of Theorem Proving Procedures"
    - In 1982 he received the Turing Award
- Given: Propositional logic formula F
- Sought: Is F satisfiable? I.e., are there variable values from {0, 1} for which F is 1?

- Proof consists of two parts
  - SAT $\in$ NP (not so complicated)
    - Principle: NTM "guesses" solution and checks its correctness (in polynomial time)
  - SAT is NP-hard (more difficult…)
  - For details see literature

Consequences of SAT being NP-complete

- **Any problem in NP is polynomial-time reducible to SAT**: $X \leq_p \text{SAT}, X \in \text{NP}$
- Deterministic algorithms for solving SAT have exponential complexity $2^{O(n)}$
  - typical brute-force solution: try all variable combinations
  - this results in an upper bound for the complexity of all problems in NP of $2^{p(n)}$
    - p(n) is a polynomial

- Note:
  - We consider decision problems here
    - i.e., problems that can be answered by “yes” or “no”
  - Finding the actual solution can be even more difficult

Consequences of SAT being NP-complete

- We can now easily show that other problems X are also NP-complete: Reduce SAT to X
  - SAT $\leq_p$ X (= solve SAT by transforming its input to X, then solve X)
  - X is at least as hard as SAT, but still in NP (it's a polynomial-time transformation!)
- Several thousand NP-complete problems are known, from various domains
- A selection can be found, e.g., here: http://en.wikipedia.org/wiki/List_of_NP-complete_problems

- If you find an algorithm with polynomial time complexity for any one of them, then you
  - automatically have a polynomial algorithm for all problems in NP
  - have proven that P = NP
- If you can show of any one of them that it is not in P, then
  - none of them is in P
  - and we have P ≠ NP

Karp‘s NP-complete Problems

- SAT
  - Clique
    - Set Packing
    - Vertex Cover
      - Set Cover
      - Directed Hamiltonian Cycle
        - Undirected Hamiltonian Cycle
  - 0-1 Integer Programming
  - 3-SAT
    - Graph Coloring
      - Partition into Cliques
      - exact Cover
        - Knapsack
          - Partition
            - Maximum Cut

Some of the 21 NP-complete problems and their reductions
by R. Karp, 1972: „Reducibility Among Combinatorial Problems“

3-SAT

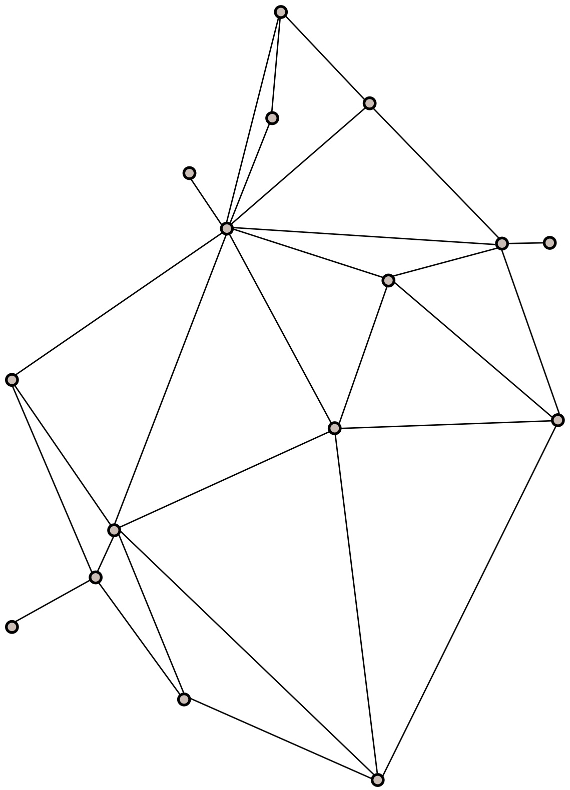
- Restriction of SAT
- Given: Propositional logical formula F in conjunctive normal form (CNF) with a maximum of 3 variables per term
- Sought: Is F satisfiable? I.e., are there variable values from {0, 1} for which F is 1?
- It can be shown: SAT $\leq_p$ 3-SAT $\longrightarrow$ 3-SAT is NP-complete

- Notes:
  - any logical formula formula can be transformed into CNF
  - however, this requires exponential effort, and we need a polynomial reduction
  - Luckily, polynomial reduction does not require exact equivalence,
  - but only: if F can be fulfilled, then the transformed formula F' can also be fulfilled (and vice versa)
- all k-SAT problems with $k \geq 3$ are NP-complete
- 2-SAT, on the other hand, is in P

Map Coloring

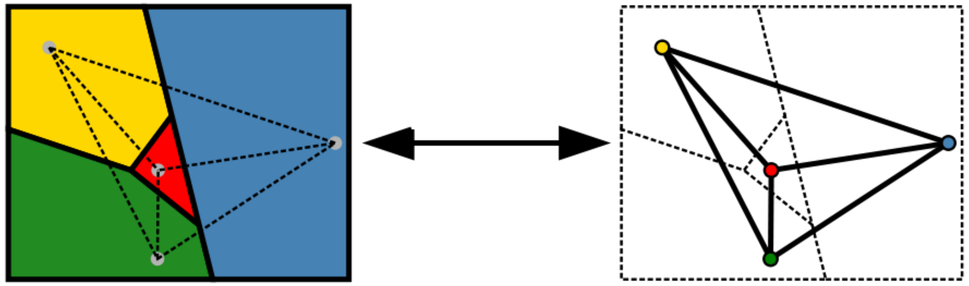
Is it possible to color a map with k colors in such a way that neighboring countries always have different colors?

Graph Coloring

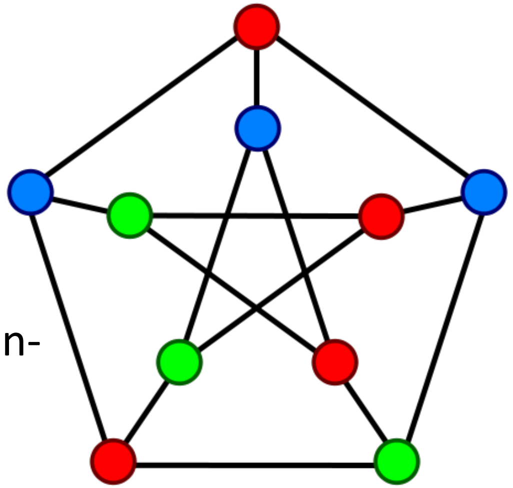


This is the graph coloring (*Graphfärbung*) problem:

- the vertices are colored,
- the edges define neighborhood.



We can do this for non-planar graphs as well



Graph Coloring

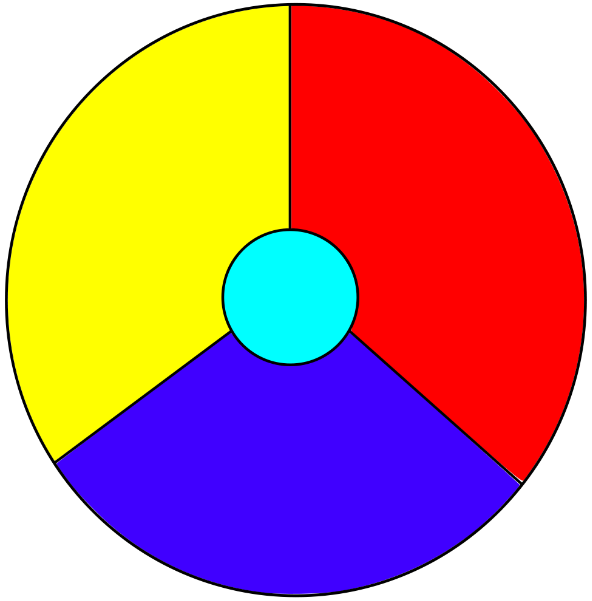
- General graphs
  - k-coloring for $k \geq 3$ is NP-complete
  - 2-coloring is in P
- Planar graphs
  - 2-coloring is in P
  - 3-coloring is in NP-complete
  - k-coloring for $k \geq 4$ has **constant runtime**!
- Reminder:
  We consider the decision problem:
  Can the graph be colored using k colors?

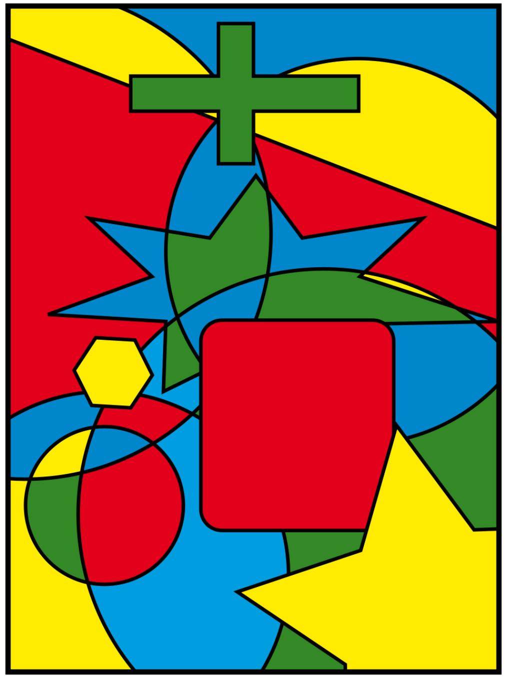
The Four Color Theorem

- 4 colors are always sufficient to color a planar graph (map)
- Assumption existed since 1852
- One of the first problems proven with the help of a computer system (1976)
- A formal proof with the help of a theorem prover followed in 2004

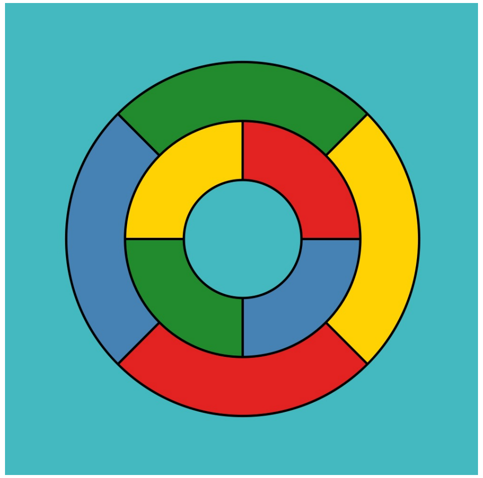
Four Color Theorem

Technische Hochschule Rosenheim

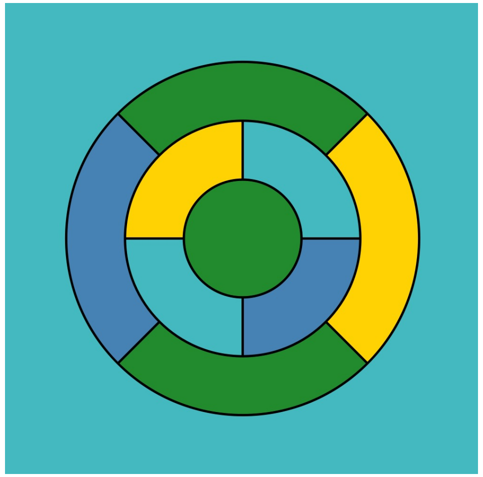




© Inductiveload, Four Colour Map Example, CC BY-SA 3.0



Coloring using 5 colors…



…but 4 suffice.

Graph Coloring – Applications

In addition to coloring maps, many other applications

- Scheduling
  - Scheduling processes in operating systems
  - Assignment of aircraft to flights
  - Allocation of bandwidth to radio/television stations, mobile communications, …
  - Creating class schedules (with constraints regarding rooms, students, teachers)
- Compilers
  - which variable values are kept in registers?
- Sudoku
  - special graph, 81 nodes, 9 colors

Travelling Salesman Problem (TSP)

- Given: n cities, as well as the distances (km, time, cost, ...) in between
- Sought: Which sequence of cities is the shortest round trip?
  - all cities should be included exactly once
  - as a decision problem: Is there a round trip with a length smaller than a given constant k?

- Corresponds to Hamilton circles in graphs
  - each city is a vertex
  - every connection between cities is an edge
  - the distance corresponds to an edge weight

Travelling Salesman Problem (TSP)

- TSP (decision problem) is NP-complete
  - the time complexity of the naïve solution is even $O(n!)$
    - this would not be in NP; remember: an upper bound for the complexity of all problems in NP is $O(2^{p(n)})$
  - good algorithms reduce this to $O(2^n)$
- TSP (actual solution) is NP-hard

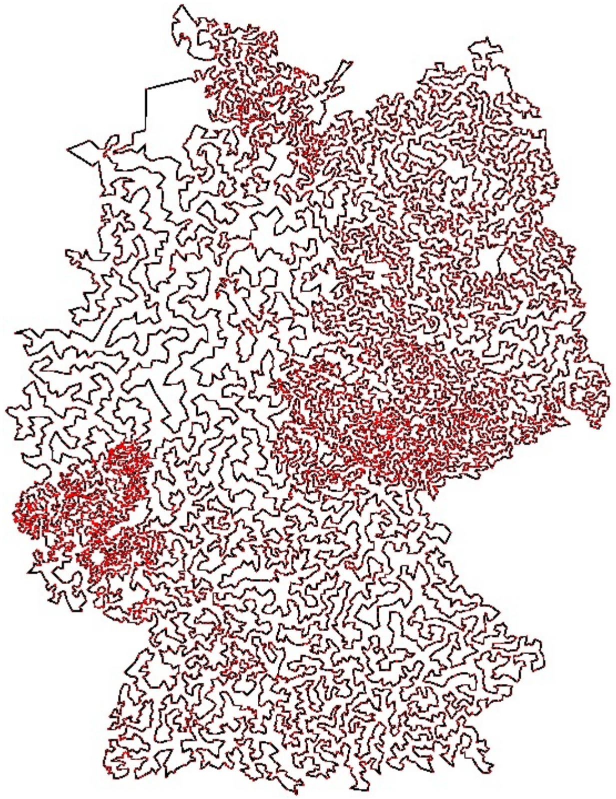
- How hard it is $O(n!)$?
  - suppose you need a computation time of 1 second for the shortest round trip through 10 cities
  - then, for 20 cities, 670 442 572 800 seconds are needed = 21259 **years**

TSP – Example

- Round trip through the 15 largest cities in Germany
- There are 14! / 2 different round trips
  - 14! / 2 = 43 589 145 600
- The one shown is the shortest round trip

TSP – Examples

- Round trip through 15,112 German cities (2001)
  - Use of 110 CPUs
  - equivalent computing time (500MHz Alpha CPU): **22.6 years**
- Round trip through 24,978 Swedish cities (2004)
  - Length: 72,500 km
  - Linux-Cluster with 96 Intel Xeon 2.8GHz CPUs (dual core)
  - equivalent computing time (2.8GHz dual core Xeon): **84.8 years**
- Layout of electronic circuits
  - 85,900 nodes (2005/06) – the current record for TSP
  - equivalent computing time (2.4GHz AMD Opteron): **136 years**



(c) https://www.math.uwaterloo.ca/tsp/gallery/itours/d15112.html

- For more examples & data see: http://www.math.uwaterloo.ca/tsp/optimal/index.html

What to do in Practice?

- We do not have any efficient algorithms to find optimal solutions to problems like TSP or creating class schedules
  - Highly likely, these do not exist (assumption: P ≠ NP)
  - Note: Even if P = NP, we would not have an efficient algorithm for round trips, as we are not interested in the decision problem but rather the actual round trip (which is NP-hard but not NPC)

- In practice: Find approximations using optimization algorithms
  - these find suboptimal solutions, but much faster
  - based, e.g., on (non-linear) numerical optimization, statistics, or probabilistic algorithms
  - in some cases, like TSP, we can establish lower bounds and give estimates on the solution's quality

TSP-Approximations – Example: World TSP

- 1,904,711 city locations throughout the world
- Best tour found so far (15 Feb 2021): 7,515,755,956 m
  - not starting from scratch,
  - but based on a tour of length 7,515,767,286 m on 11 Feb 2021
  - difference: 11,330 m
  - this is considered a huge improvement!
- Improvement between Oct 2011 and June 2020: 7,604 m
- A lower bound for the tour is 7,512,218,268 m
  - difference to best solution so far: 3,549,018 m (= 0.0471%)

(c) https://www.math.uwaterloo.ca/tsp/world/index.html

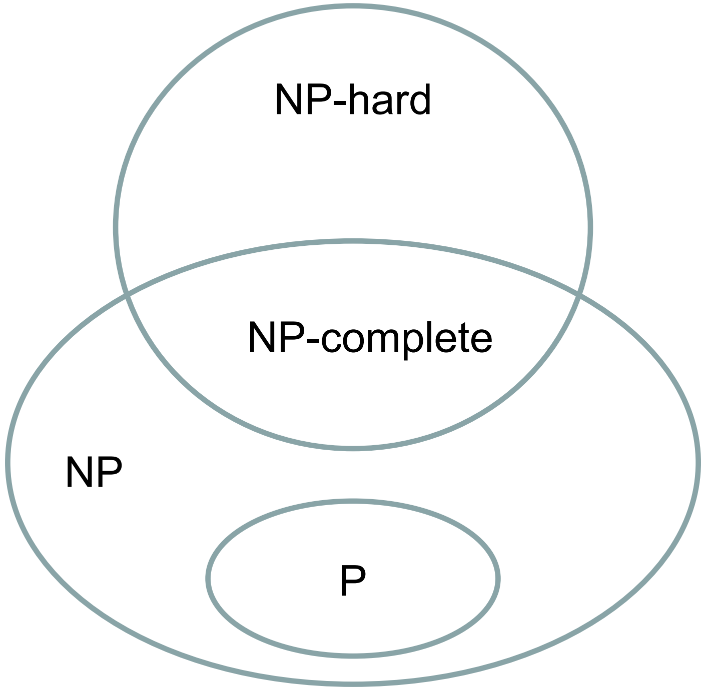
- Details & data see: https://www.math.uwaterloo.ca/tsp/world/index.html

Other Problem Classes

NP-hard Problems Outside of NP

- Proof that problems lie in NP is not possible here
- so these are even more difficult than NP-complete problems

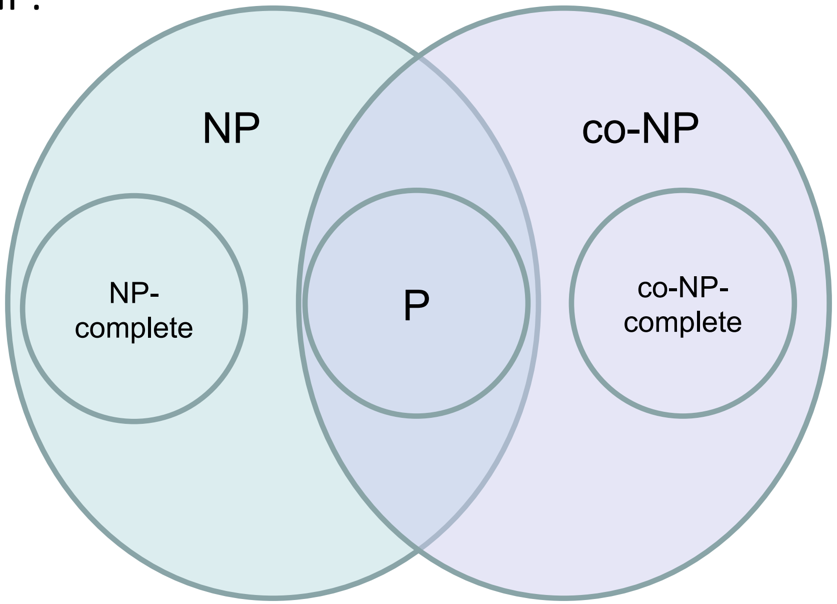
- Examples:
  - Word problem for type-1 languages
  - Inequivalence of regular expressions
    - and thus: for regular grammars or nondeterministic finite automata
    - Note: Equivalence of deterministic finite automata is in P
      - Conversion nondeterministic ⟶ deterministic requires construction of the power set
      - and thus has exponential complexity



Assumption: P ≠ NP

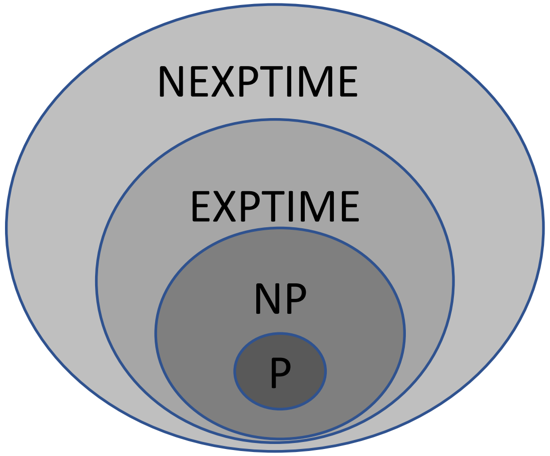
co-NP

- co-NP: Set of decision problems whose complements are contained in NP
- Example: The PRIMES problem
  - „Is a number prime?“ is in NP
  - „Is a number not prime (= composed)? “ is in co-NP
- Since P is closed with regard to complement, we do know for sure: P = co-P
- General belief: NP ≠ co-NP
  - if it can be proven for any NP-complete problem that it is in both, NP and co-NP: NP = co-NP
  - so far none has been found, hence the general belief
  - in the case of P = NP: NP = co-NP holds
- By the way, PRIMES is in NP and co-NP
  - this is a very strong indication that a problem is not NP-complete
  - and in fact, PRIMES is in P
  - a polynomial algorithm was published in 2002



EXPTIME and NEXPTIME

- EXPTIME
  - Set of all decision problems that can be solved by a deterministic TM in time $O(2^{p(n)})$
    - p(n) is a polynomial
  - there are EXPTIME-complete problems, e.g.,
    - modified halting problem: Does a deterministic TM halt after at most k steps?
    - Position analysis for generalized chess, checkers, go (arbitrary number of pieces on arbitrary sized board)
- NEXPTIME
  - corresponding class for nondeterministic TM
- Remarks
  - if P = NP, then EXPTIME = NEXPTIME
  - But: P $\subsetneq$ EXPTIME and NP $\subsetneq$ NEXPTIME



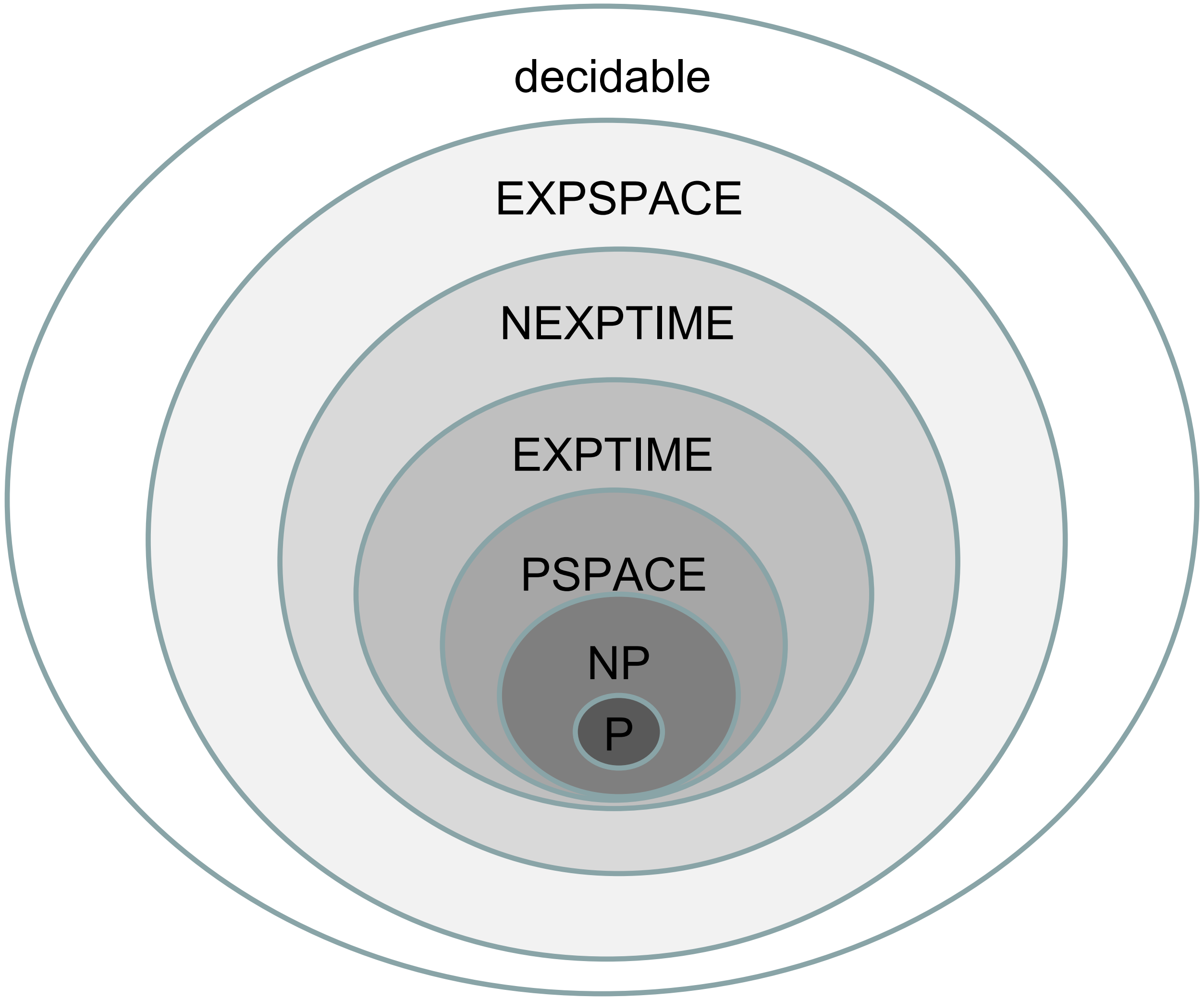
PSPACE and NPSPACE

- PSPACE
  - Set of all decision problems that can be solved by a deterministic TM with polynomial space
- NPSPACE
  - corresponding class for nondeterministic TM
- Obviously: P $\subseteq$ PSPACE and NP $\subseteq$ NPSPACE
  - a TM can write at most polynomial many symbols to the tape in a polynomial number of steps (time)
- It can be proven: PSPACE = NPSPACE
- There are PSPACE-complete problems, e.g.,
  - Word problem for type-1 languages
  - Satisfiability of Boolean formulas with quantifiers ($\forall$, $\exists$)

EXPSPACE and NEXPSPACE

- EXPSPACE
  - Set of all decision problems that can be solved by a deterministic TM with space $O(2^{p(n)})$
    - p(n) is a polynomial
- NEXPSPACE
  - corresponding class for nondeterministic TM
- It holds
  - EXPSPACE = NEXPSPACE
  - PSPACE $\subsetneq$ EXPSPACE
  - EXPTIME $\subseteq$ EXPSPACE (likely: EXPTIME $\subsetneq$ EXPSPACE)
- There are EXPSPACE-complete problems, e.g.,
  - Do two given regular expressions define different languages?

Complexity Classes – Overview



And there are more..., e.g.,

- for probabilistic algorithms
- below P
- for quantum computers
- to consider the calculation of a solution instead of the decision problem (the functional problem)

Summary

- Order of complexity: O-Notation
  - is asymptotic
  - efficiency: separation between polynomial and exponential complexity
  - in practice, it will already get hard from approx. $O(n^4)$
- Complexity Classes
  - P: Decision problems that can be solved by a deterministic TM in polynomial time
  - NP: as P for deterministic TM $\longrightarrow O(2^{p(n)})$ for deterministic algorithms
- NP-completeness
  - Problems that are NP-hard and completely contained in NP
  - They are all connected by polynomial-time reduction
  - Whether P = NP is one of the great unsolved problems of computer science
  - Belief, based on many indications: P ≠ NP
  - There are a lot of NP-complete problems with practical relevance

Sources

- H. Ernst, J. Schmidt und G. Beneken: *Grundkurs Informatik*. Springer Vieweg, 7. Aufl., 2020.
- Schöning, U.: *Theoretische Informatik – kurz gefasst*. Spektrum Akad. Verlag (2008)
- Hopcroft, J.E., Motwani, R. und Ullmann, J.D.: *Einführung in die Automatentheorie, formalen Sprachen und Komplexitätstheorie*. Pearson Studium (2002)
- Sedgewick, R.: *Algorithmen in C++*, Addison-Wesley (1992)