Foundation of Information Processing

Complexity of algorithms

Considerations about feasible algorithms

When an algorithm is feasible
(including, solvable computationally)?

How algorithms can be compared?

Are there tasks which do have no feasible algorithm?
Thus, it is not possible to develop any feasible algorithm.

Estimation of the efficiency of algorithms

Algorithms are **different**, depending on a task and a selected strategy of problem solving.

Problem solving strategies affect **computation time**, **memory requirements**, and, for example, in case of approximate algorithms also affect the **accuracy** of the result.

Algorithms can be **compared** based on these criteria.

Complexity

”For every complex problem there is an answer
that is clear, simple, and wrong.”
(Henry Louis Mencken, an American journalist, satirist, social critic, cynic, and freethinker, the "American Nietzsche“)

- Concepts: algorithm/task, input size.
- Criteria of performance: time and space complexity.
- Orders of complexity: polynomial vs. exponential.
- Examples of problems and their performance.
- More details given in the course Data structures and algorithms.

Source (modified from): J. Boberg, Johdatus tietojenkäsittelytieteeseen, Turun yliopisto, 2010 (in Finnish)

What is complexity?

- Complexity of an algorithm:
  - Resources needed to execute an algorithm as a function of the size of the task *in the worst case*.
- Complexity of a task:
  - Resources needed to execute *the best algorithm* for a task as a function of the size of the task.
- Size of the task:
  - The size of a given input.
  - Example: if the complexity of a problem is defined by $f(x) = x^2$ then x is the input.
  - When the input increases more computation is needed.
- The main concern is that **how solvable** (feasible) is the **solution** when the **input increases**, and whether there are other solutions available.

How to measure the complexity?

- Measuring the performance of algorithms and what can be seen from these criteria.
- Time complexity *T(n)*:
  - How much time is needed for the execution of the algorithm?
- Space complexity *S(n)*:
  - How much space (memory) is needed?
- Hardware requirement:
  - Physical devices, e.g., parallel computing.
  - This course focuses on the time and space complexity.
- Orders of the complexities of algorithms.
  - What kinds of orders *T(n)* and *S(n)* can be?
  - The orders can be defined using mathematical functions.

Orders of the complexity

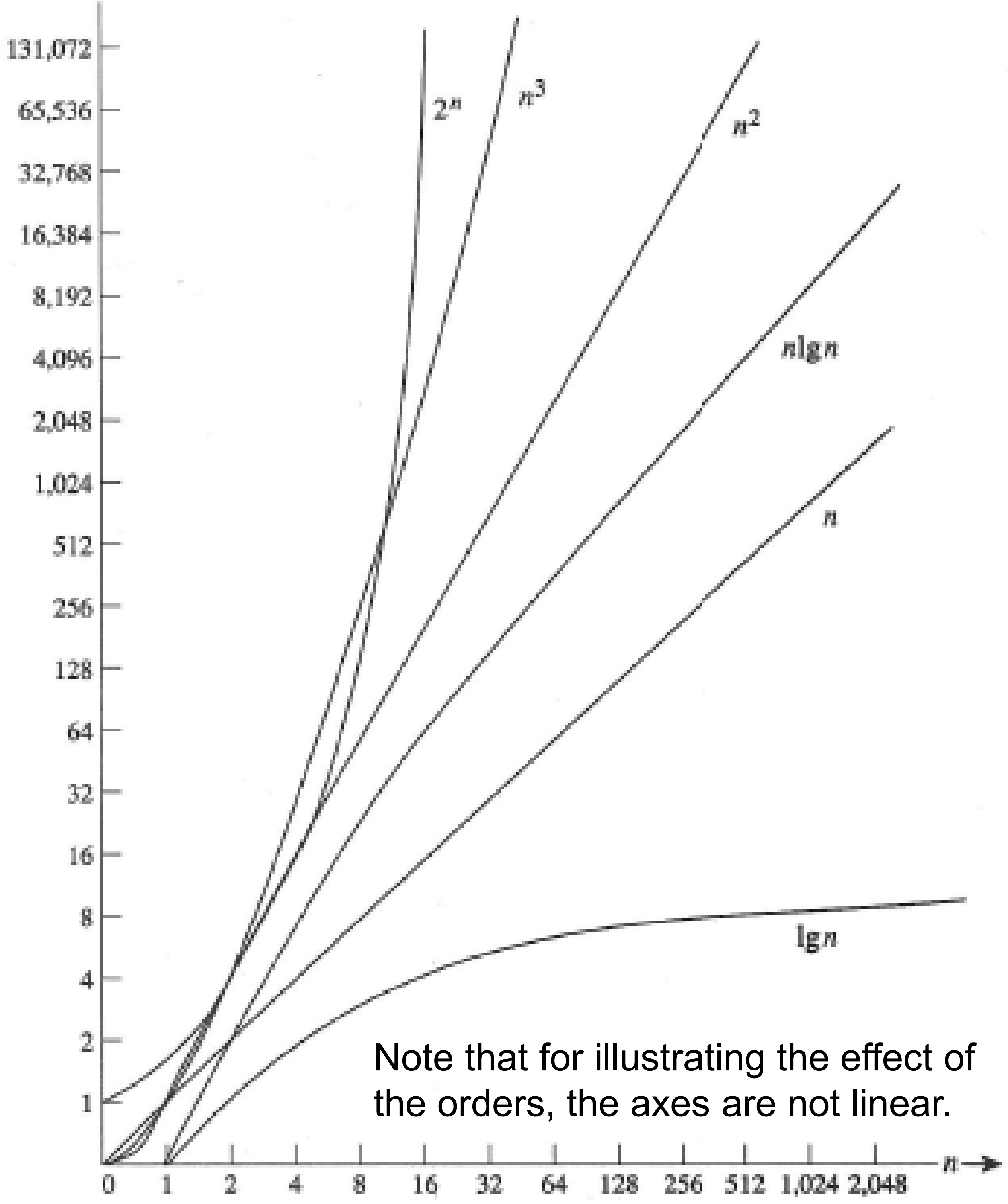
Constant:
$T(n) \sim 1$
(independent from the input)

Logarithmic:
$T(n) \sim \log_c n$
(in the figure lgn)

Linear:
$T(n) \sim n$

Polynomial:
$T(n) \sim n^c$

Exponential:
$T(n) \sim c^n$



Note that for illustrating the effect of the orders, the axes are not linear.

Feasible algorithms and infeasible algorithms

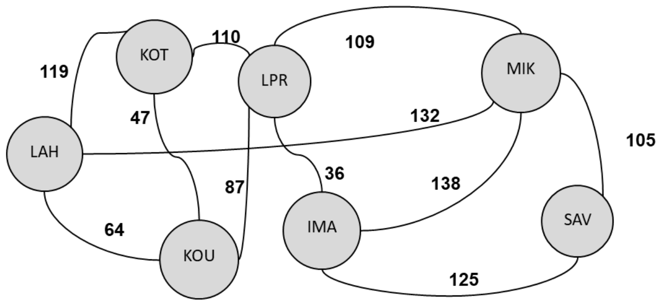
- Where is the border for an algorithm to be solvable or not?
- Algorithms with the **exponential** complexity **cannot be computed** in practice.
- This can be seen in the following numerical computations (also in the figure of the orders on the previous page):

$\log_2 n$	n	n^2	$n^2 + 5n + 3$	n^3	2^n	n!
0	1	1	9	1	2	1
n. 3	10	100	153	10^3	1024	3628800
n. 7	100	10 000	10 503	10^6	$\sim 10^{30}$	$\sim 10^{158}$
n. 10	1 000	1 000 000	1 005 003	10^9	...	...
n. 13	10 000	100 000 000	100 050 003	10^{12}		
n. 17	100 000	10 000 000 000	10 000 500 003	...		
n. 20	1 000 000	1 000 000 000 000	1 000 005 000 003			

Options for optimal and complete solutions

Approximate algorithms:

- Error in problem solving is accepted (preferably *the known maximum error*) to obtain a fast enough efficient solution.
- The goal is to make a feasible algorithm.
- **Example**: **travelling salesperson's problem**.
  - Towns are connected by roads.
  - Each town can be connected to another town directly, or via other towns by one or many routes.
  - A travelling salesperson should visit all towns so that the length of traveling is as short as possible.
  - The optimal solution is exponential.
    - n*(n-1)*(n-2)*…* 2*1 = n! routes in case of n towns.
    - The solution using dynamic programming is in the oder of 2^n.
  - The solution can be found in polynomial time when the error of 0%-50% is accepted as compared to the optimal solution.



Options for optimal and complete solutions

Randomized algorithms:

- **Example**: deadlock of processes
  - Process A needs to have the both resources X and Y at the same time be able to proceed in the execution.
  - However, Process B has the same requirement.
  - If each of them has one resource (X or Y) only they need to wait to get the other resource. Thus, they have locked each other (deadlock).
- Solution: Let us *allocate* resources *at random* so one of them could get the both resources at the same time. In practice, *flip a coin* in resource sharing between A and B so long as one of them gets the both resources.
- The risk caused by random selection must be admitted and accepted.
- Fortunately, in this case it is not so probable to fail in a longer sequence of coin flipping.
  - The probability to allocate one resource only per one process is relative to $1/2^n$ where n is the number of throws.

Options for optimal and complete solutions

Randomized algorithms:

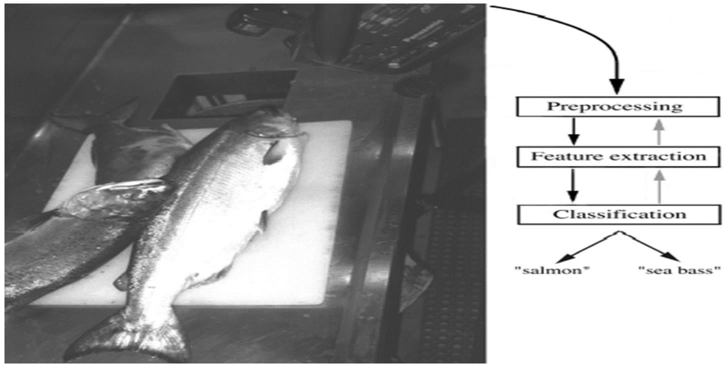
- **Another case**:

  How many times a dice must be thrown to get the face “six”?
- It is not possible to define the exact number of throws needed.
- However, it is also here possible to estimate the probability of success (or failure) as a function of the number of throws.
- The success to get the first “6”:
  - 1 throw: 1/6. (lucky immediately)
  - 2 throws: 5/6 x 1/6. (the 2nd throw successful)
  - 3 throws: 5/6 x 5/6 x 1/6. (the 3rd throw successful)
  - Etc.
- It is also possible to estimate the failure to get “6” as a function of the number of throws.

Options for optimal and complete solutions

Heuristic algorithms (rules of thumb).

- Example: sorting types of fish.
  - If a fish is big enough, it is salmon.
  - This rule is useful only if there are enough salmons among big fish types.
    - What is “enough”?
  - Moreover, it must be estimated what is the risk that all other bigger fish types always are misclassified.
    - Problems in cooking?



Source: Duda, Hart & Stork: Pattern Classification, 2001.

How to define the complexity of an algorithm?

- It is very important to know, especially,
  - the **time complexity** $T(n)$ of an algorithm.
- Thus, the complexity of each algorithm should be defined.
- Let us consider this matter by the following examples:
  - Travelling salesperson (already considered).
  - Sorting algorithms.
  - Tower of Hanoi.
- The goal of this course is not to learn how to prove the complexities mathematically but to get very first experience how to “observe” different complexities.
- More details about proving the complexities are given in the course Data structures and algorithms.
- Next, let us consider the examples.

Complexity of sorting algorithms

Exchange sort:

- Problem: sort the given elements in the list in the increasing order.
- Straightforward solution: compare each element to all other elements.

```
MODULE exchangesort(T)
    FOR i := 1, 2, ..., n DO
        FOR k:= 1, 2, ..., n DO
            IF T[i] > T[k] THEN swap(T[i], T[k]) ENDIF
        ENDFOR
    ENDFOR
ENDMODULE
```

- The repetition structure FOR (loop) is nested with another loop.
- The both FOR structures are executed n times where n is the number of the elements => n times n.
- The complexity is in the order of $n^2 \Rightarrow T(n)$ is polynomial.
- Could the algorithm be improved?

Complexity of sorting algorithms (cont.)

Insertion sort:

- Solution: let us not compare all the elements to each other but only those elements which have not been ordered yet.

```
MODULE insertionsort(T)
    FOR i := 1, 2, ..., T.length–1 DO
        FOR k:= i+1, ..., T.length DO
            IF T[i] > T[k] THEN swap(T[i], T[k]) ENDIF
        ENDFOR
    ENDFOR
ENDMODULE
```

- Complexity: $T(n) = n(n-1)/2 \sim n^2$.
  - Let us denote T.length as n.
  - It is needed $n - 1$ iterations where there are $n - i$ comparisons.
  - Thus, $(n-1) + (n-2) + \cdots + 2 + 1 = n(n-1)/2$ comparisons.

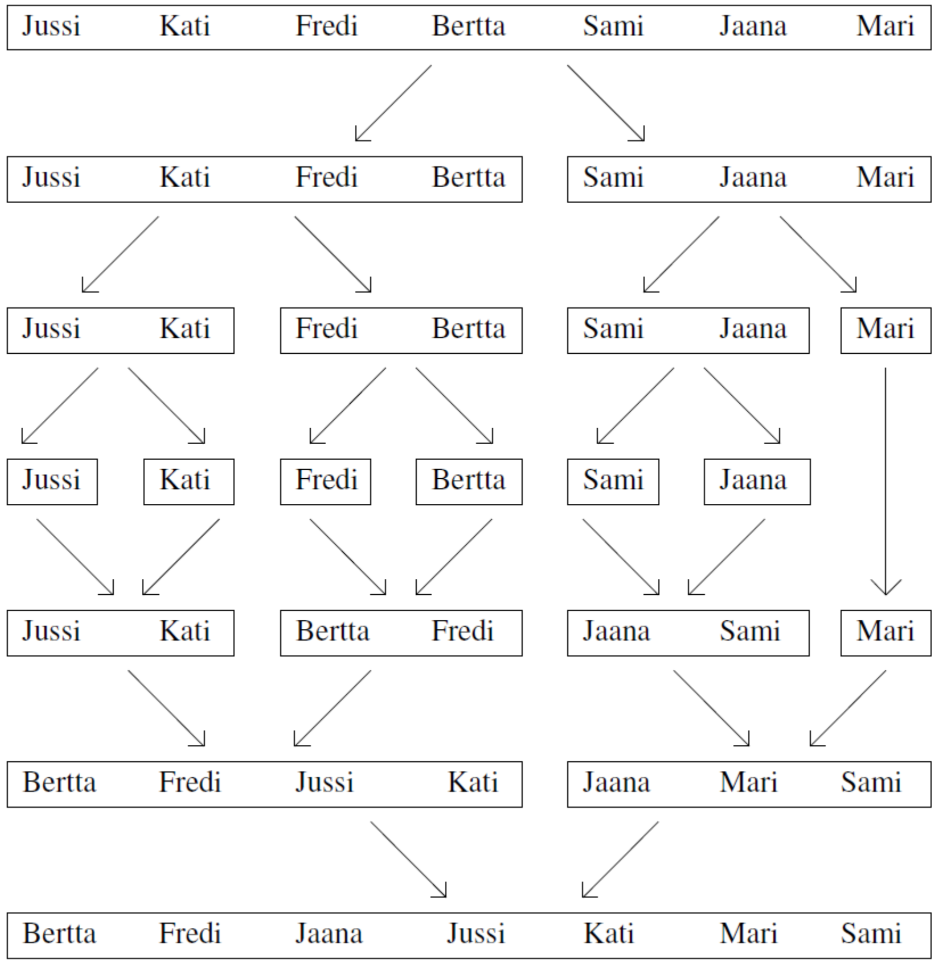
Complexity of sorting algorithms (cont.)

Merge sort:

- A tree structure is for more efficient sorting due to a logarithmic nature
- The root => child nodes => grandchild nodes => etc.
- Divide the list into two halves recursively until there is one element only, and then merge the sub lists back to one list.
- Recursive algorithm:
  $$T(n) = T(n/2) + T(n/2) + n$$
  $$= 2T(n/2) + n$$
- $T(n) \sim n \log_2 n.$
- $n \log_2 n$ vs. $n(n-1)/2.$

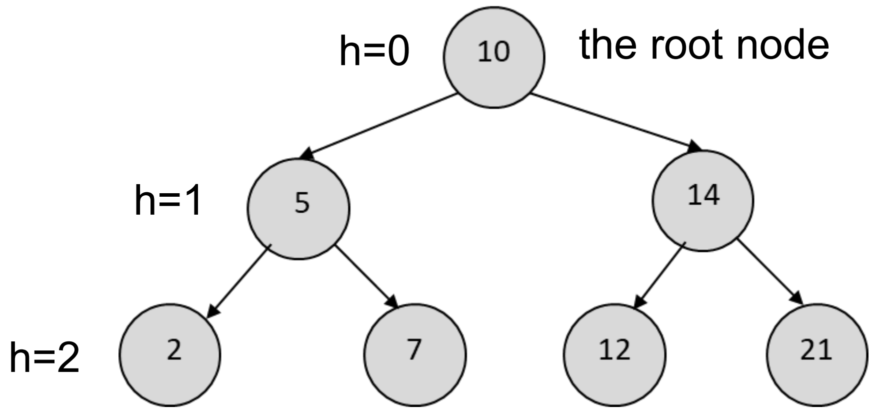
n	$n \log_2 n$	$n(n-1)/2$
16	64	120
32	160	496
64	384	2016
128	896	8128

Example: sort the following Finnish given names in the alphabetical order.



A binary tree: logarithmic complexity in search

- A tree is an **ordered binary tree**, if each parent node has at maximum two child nodes so that
  - the value of the parent node is *greater* than the value of *the left child node*, and
  - the value of the parent node is *smaller* than *the right child node*.
- **Example**: What is the complexity to find a given value in an ordered binary tree as a function of the number of nodes? In the worst case the value is to be found at the furthest height from the root node or not found at all.



Height h	Nodes at Height h	Nodes in total n
0	1	1
1	2	3
2	4	7
3	8	15
4	16	31
…	…	…

- $n = 2^{h+1} - 1$ for the full binary tree at the height h.
- $n + 1 = 2^{h+1} \Rightarrow \log_2(n+1) = \log_2(2^{h+1}) \Rightarrow\ h = log_2(n+1) - 1\ \sim \boldsymbol{\log_2 n}$
- Thus, the search in a binary tree is logarithmic as a function of the input n.

Tower of Hanoi: exponential complexity

Tower of Hanoi:

- Move n disks stacked on the start rod in order of decreasing size onto the goal rod so that a larger disk is always lower than a smaller disk while making moves.
- Recursive algorithm:

$$T(n) = T(n-1) + 1 + T(n-1)$$
$$= 2T(n-1) + 1.$$

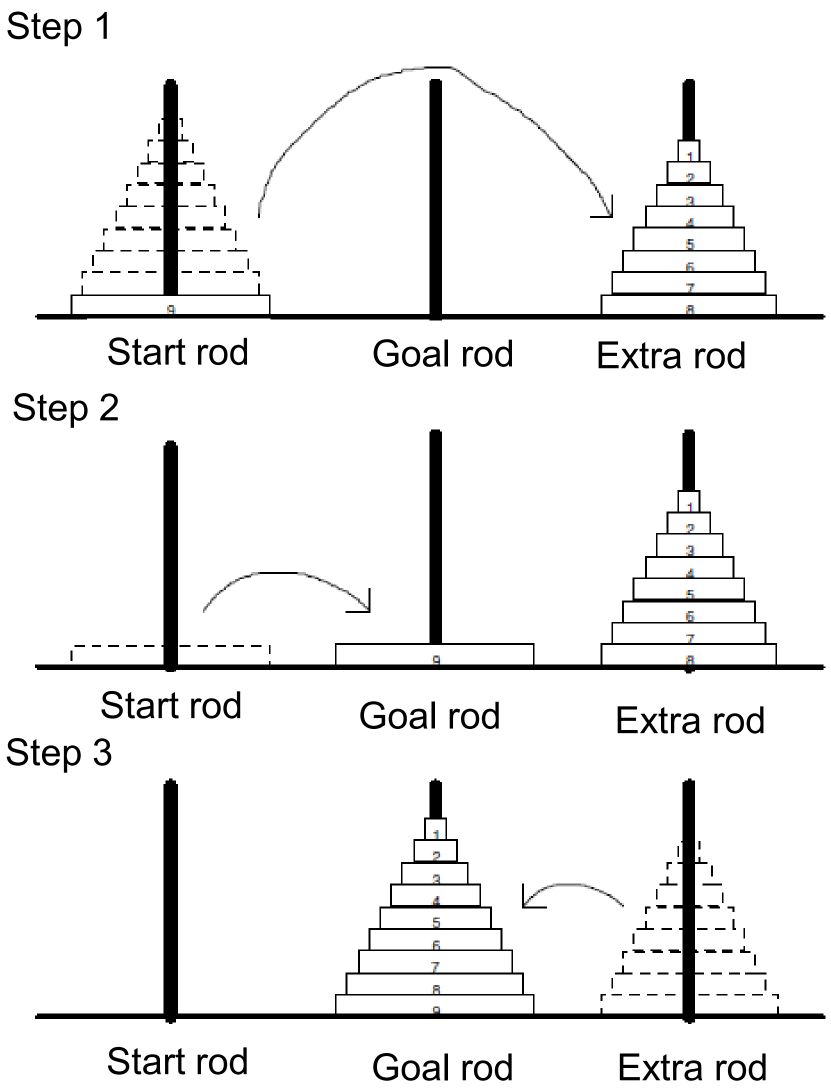
- The complexity is exponential:

$$T(n) \sim 2^n - 1.$$

=> The solution is infeasible in practice.

8	255
16	65535
32	4,29E+09
64	1,84E+19
128	3,4E+38

64 discs &
1 s/move =>
$2^{64} - 1\ s$
$\Rightarrow$ 585 billion years!
(US billion)



Sets of problems P and NP

- Let ***P*** represent a set of those problems which can be solved by
  - a deterministic polynomial algorithm.
  - Thus, the feasible solution exists such as for the problem of sorting.
- Let ***NP*** (nondeterministic polynomial problems) represent a set of those problems which can be solved by
  - a nondeterministic algorithm in polynomial time.
  - Thus, there is no feasible solution (or not yet found) such as for the problems of packing backpacks and travelling salesperson.
  - These problems can be solved in polynomial time approximately with the risk of less exact results than in the optimal solution.
  - In practice, the optimal solution cannot be computed.
- One of the famous big questions in Computer Science: naturally, $P \subseteq NP$, but it is not known whether $P=NP$, or is NP the genuine subset of P?

Summary

- There are problems which can be solved using a **feasible algorithm**, i.e., the problem is solvable in a reasonable time.
- The analysis of the **complexity** (orders of computations needed) enables comparisons of performance of algorithms, and thus, suitable solutions can be reviewed and selected.
- If the optimal (exact) solution cannot be computed in practice (so no feasible algorithm exists) an **approximate algorithm** can be formulated with the risk of *less exact results* than in the optimal solution.