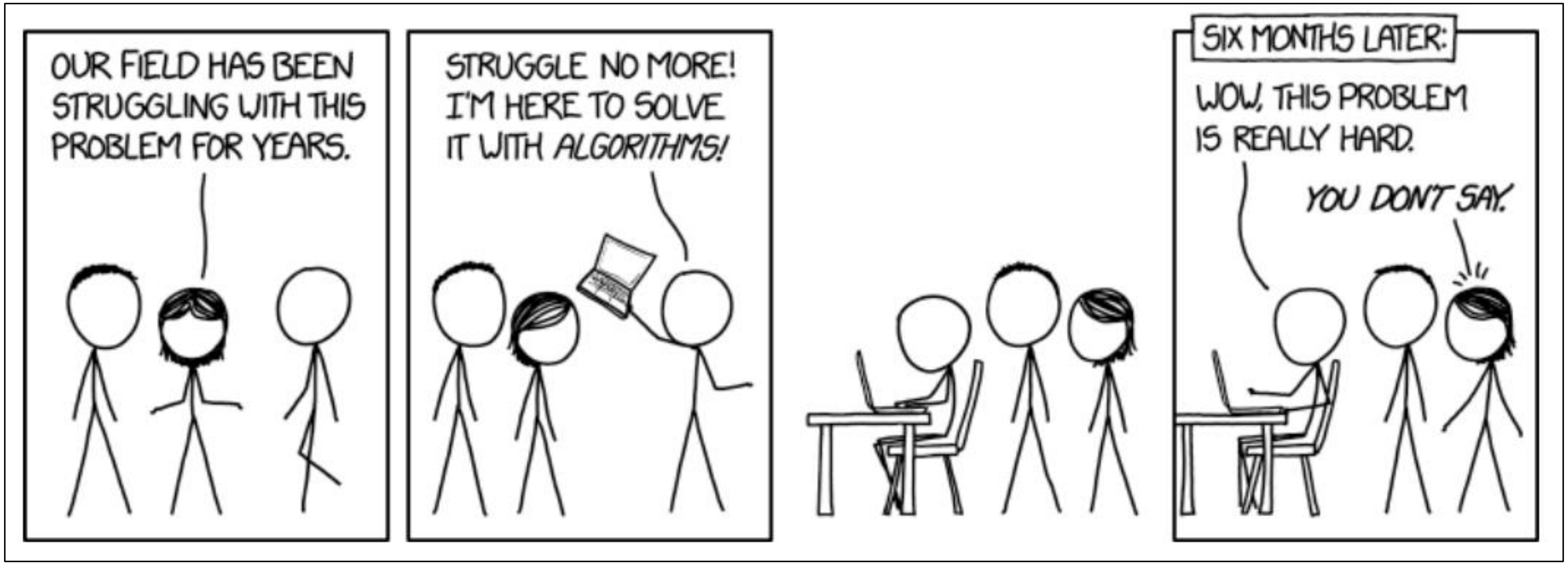
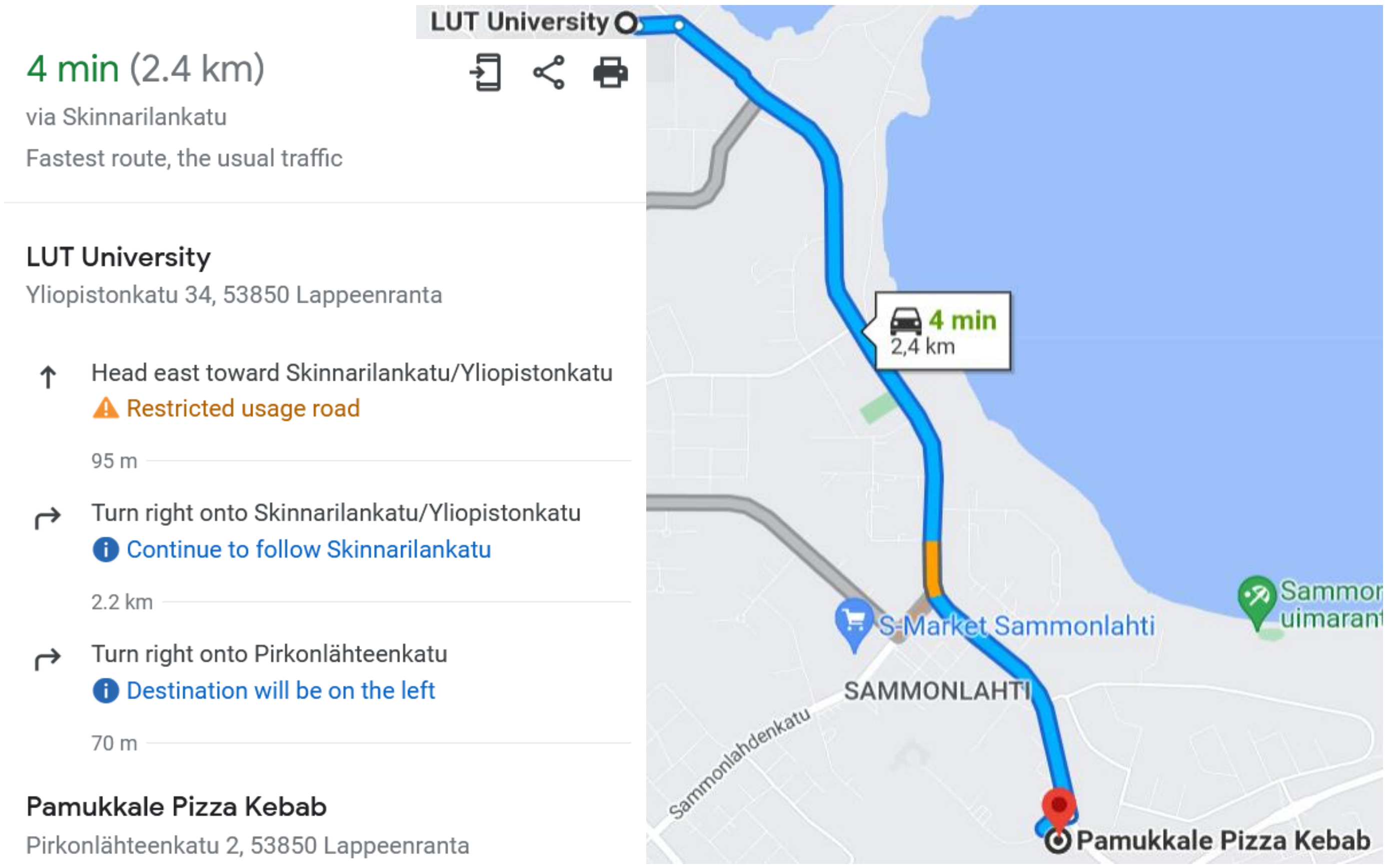
9. Algorithms


OUR FIELD HAS BEEN STRUGGLING WITH THIS PROBLEM FOR YEARS.
STRUGGLE NO MORE! I'M HERE TO SOLVE IT WITH *ALGORITHMS!*
SIX MONTHS LATER:
WOW, THIS PROBLEM IS REALLY HARD.
YOU DON'T SAY.

Everyday “algorithms”

- Are these detailed enough?



4 min (2.4 km)
via Skinnarilankatu
Fastest route, the usual traffic

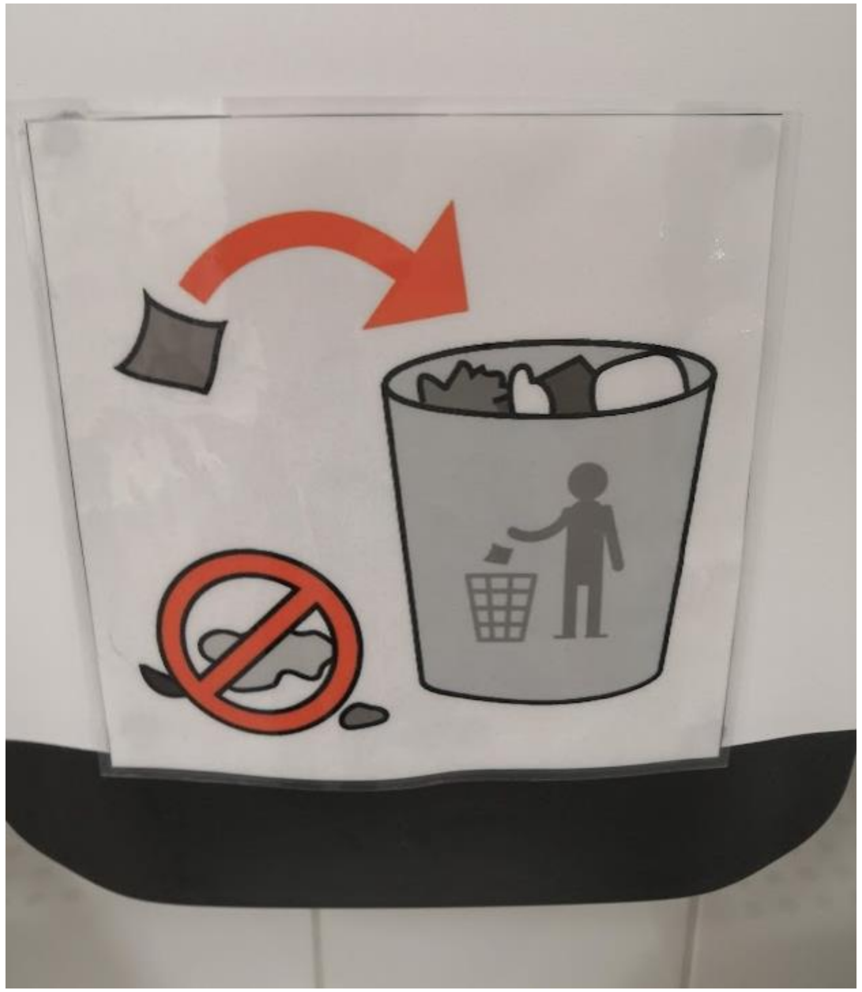
LUT University
Yliopistonkatu 34, 53850 Lappeenranta

↑ Head east toward Skinnarilankatu/Yliopistonkatu
⚠ Restricted usage road
95 m

↱ Turn right onto Skinnarilankatu/Yliopistonkatu
ⓘ Continue to follow Skinnarilankatu
2.2 km

↱ Turn right onto Pirkonlähteenkatu
ⓘ Destination will be on the left
70 m

Pamukkale Pizza Kebab
Pirkonlähteenkatu 2, 53850 Lappeenranta



What is an algorithm?

- An algorithm is basically a set of instructions
- Broadly speaking, algorithms are not limited to computing – for example, also following real-world instructions can be considered algorithms:
  - Recipes (cooking instructions of different meals)
  - Furniture assembly (instructions on how to put together a shelf)
- In these real-world applications, the detail level of instructions is usually vague (or at least related to the assumed skill level of the reader)
- Computers require the instructions in an extremely precise form
- Formal definition of an algorithm in computer science: *"An algorithm is an ordered set of unambiguous, executable steps that defines a terminating process"*
  - Termination requirement is because of the field, not because of mathematics

Church-Turing theses

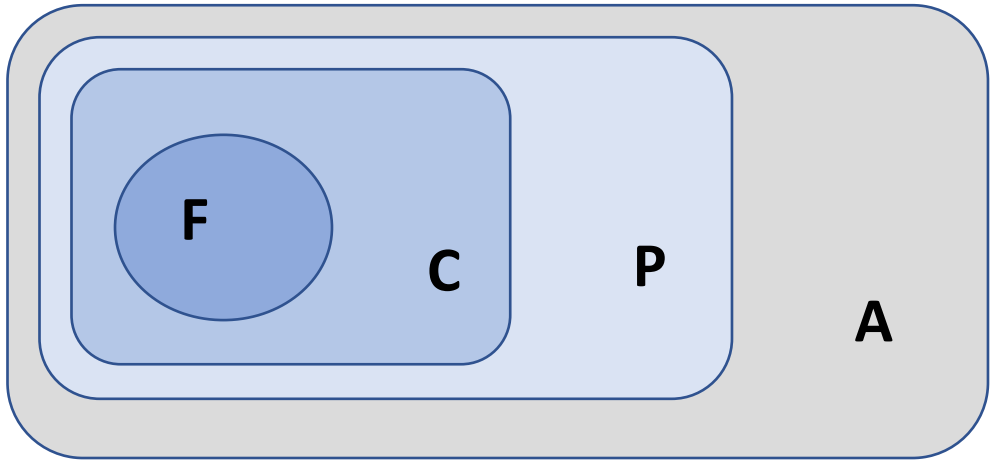
- Alonzo Church & Alan Turing proposed their own definitions for an algorithm
- Eventually, they came up with their (and some other authors') definitions being just as good, so they formulated the definition to two theses:
    1) All known reasonable definitions of an algorithm are equivalent to each other
    2) Any reasonable definition invented in the future will be equivalent to the previous ones
- First one has been proven true, second one can't be proven but is accepted true
- Consequences of the theses:
    - All algorithms can be executed by a Turing machine
    - Every algorithm that can be executed by some computer can also be executed on all other computers – i.e. all computers are equivalent to each other and can perform the same tasks!
- All computers are not created equal, though; there are major differences in efficiency and execution time

Solvability of problems

- For centuries it was thought, that all problems in the world can be solved using some algorithm
  - Solvability = has somebody discovered an algorithm for the problem or not?
- Only in the 1930s it was proven that there exists problems that *can't* be solved using an algorithm
  - David Hilbert's "Entscheidungsproblem", proven algorithmically unsolvable by Kurt Gödel's "Incompleteness Theorem" in 1931 – and soon followed by others
  - Solvability = feature of the problem itself
- So, nowadays problems can be divided to three groups:
  - Solvable problems (there exists an algorithm which solves the problem for all inputs)
  - Algorithmically unsolvable problems (proven that there is no algorithm)
  - Problems, whose solvability is (yet) unknown

Computability

- Instead of solvability, a better term would be to talk about *computability*
  - Takes into account the effectiveness of the solution: can it be found in reasonable time?
- Problems are classified based on their computability using the best algorithm:
  - Feasible problems (complexity is polynomial at max)
  - Computable problems (complexity can be exponential, but there is a finite number of options)
  - Partially computable problems (algorithm can find a solution if there is one, but can't reach a conclusion that there is no solution)
  - All problems
- Notice that smaller classes are subsets
- We'll return to these a bit later…



Analysis of algorithms

- For many problems, there are several algorithms that can be used for solving
- Especially in these cases it is natural that we are interested in ways to compare the possible algorithms to each other – i.e. analyze their performance
- Analysis of algorithms revolves around two properties:
    - *Complexity* (Note! Means calculational complexity – not "how difficult it looks like")
    - *Correctness*
- Naturally, we would want all our algorithms to be 100% correct, so more emphasis is put on the first one
- Correctness is a big criteria when we're programming the algorithm, though:
    - Advanced algorithms are often harder to prove correct
    - Decreased calculational complexity usually comes at the price of increased "visual complexity", so it also leads to increased risk of programming mistakes

Complexity

- Different problems and algorithms set various requirements for execution
- Most important computer resources are
  - Execution time (number of CPU cycles)
  - Memory use
  - System (for example, number of processors in parallel computation)
- Usually the best algorithm is considered as the one that uses the least resources
- …which in most cases translates to quickest execution time
  - In some database applications, memory use might be a limiting factor
- Demand for resources is dependent on the size of the input – denoted by n
  - Size n = length of input list / number of nodes on graph / number of cells in a table etc.

Time complexity T(n) and space complexity M(n)

- Time complexity of an algorithm is given as a function of input size – T(n)
- Unit of time complexity is not any time measure, because the execution time is very much dependent on the computer used for calculation
- Unit is therefore the number of elementary operations needed for solution

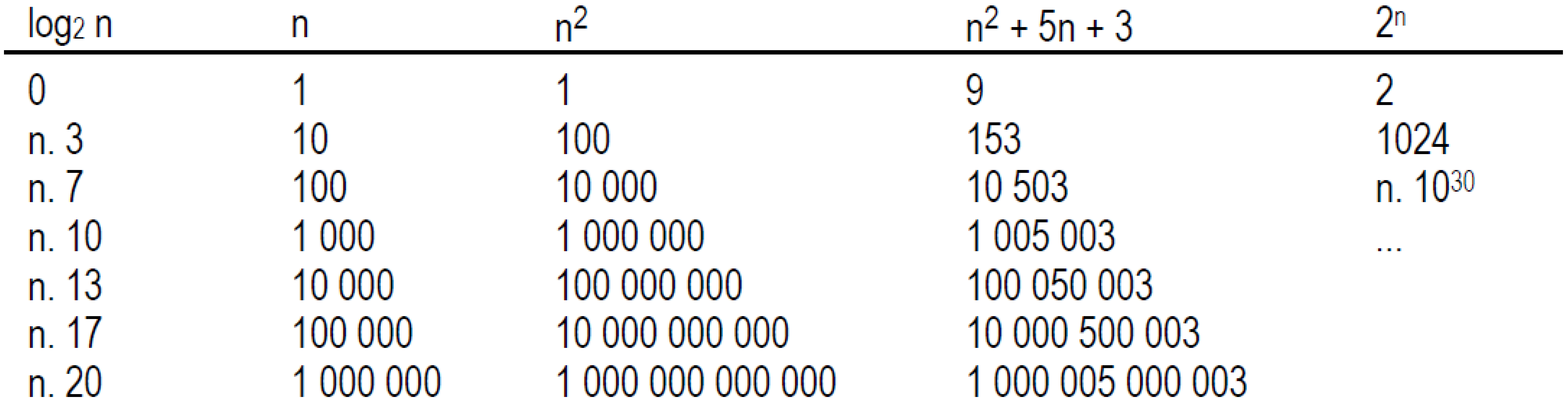
- If we are interested in the use of memory resources, the measure for this is space complexity M(n)
- Unit of space complexity is the total amount of memory space used by the algorithm

Asymptotic complexity

- The exact number of elementary operations is not interesting – it’s the order of magnitude that’s important
  - Small differences between algorithms may even out due to other reasons (data transfer etc.)
- In asymptotic complexity, only the most dominant term is considered
- Also, constant factors are ignored, because asymptotically they don’t change the situation that much
- Examples:
  - If our algorithm has time complexity $T(n) = n^2 + 5n + 3$, its asymptotic (time) complexity is simply $T(n) \sim n^2$
  - If our algorithm has time complexity $T(n) = 3n \log n + 100n - 5$, its asymptotic (time) complexity is simply $T(n) \sim n \log n$

Asymptotic complexity

- Why can we ignore the other terms? See the table below and notice how small is the difference between $T(n) = n^2$ and $T(n) = n^2 + 5n + 3$ for large input sizes



$\log_2 n$	n	n^2	$n^2 + 5n + 3$	2^n
0	1	1	9	2
n. 3	10	100	153	1024
n. 7	100	10 000	10 503	n. 10^{30}
n. 10	1 000	1 000 000	1 005 003	...
n. 13	10 000	100 000 000	100 050 003	
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	

Theta and “Big-Oh”

- The time an algorithm needs in order to find a solution is not constant even for same size inputs; it can be dependent on pure “luck”
- Therefore, when we talk about complexity, three different cases can be considered:
  - Best-case scenario
  - Average-case scenario
  - Worst-case scenario
- The actual number of elementary operations that must be performed will be something ranging from best-case to worst-case scenario
- The average-case complexity is usually referred to as theta –for example $\Theta(n^2)$
- Usually we are most interested in the worst-case scenario that gives the upper bound for complexity; this is often referred as “Big-Oh” – for example $O(n^2)$
  - NOTE! T(n) is a function – theta and Big-Oh are not!

$$\boxed{T(n) = n^3 - 4n \sim n^3 \to O(n^3)}$$

Time complexity classification

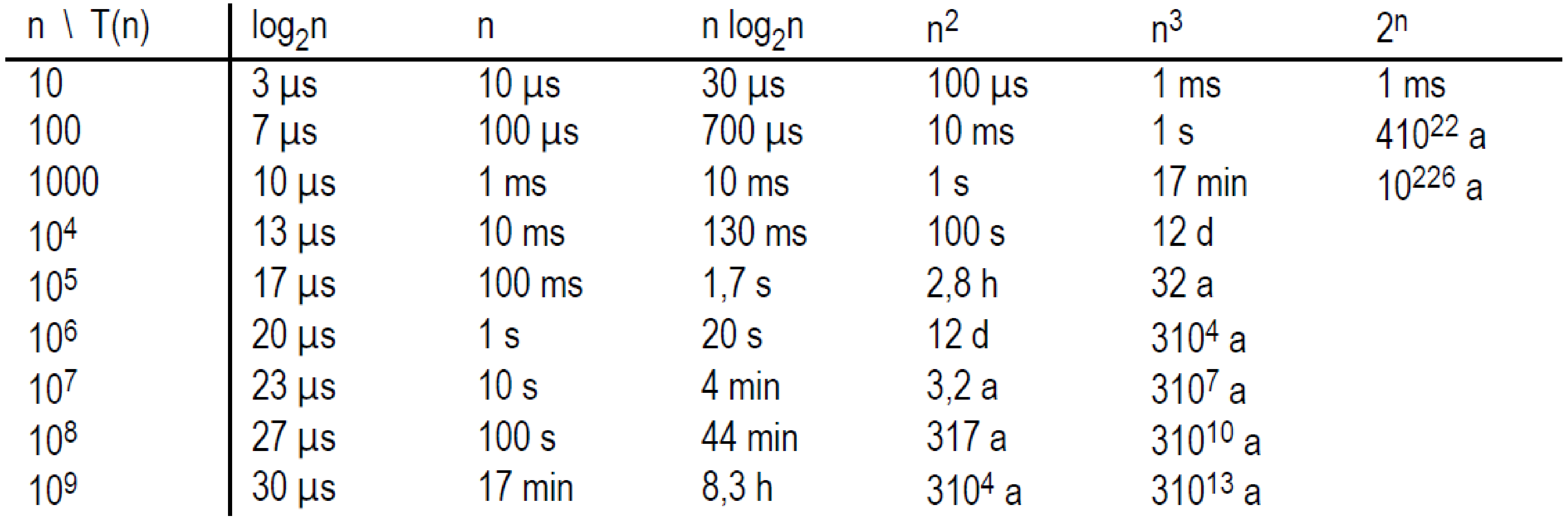
- Complexities can be classified in seven groups – from best to worst:

Constant	$T(n) \sim 1$
Logarithmic	$T(n) \sim \log n$
Linear	$T(n) \sim n$
Linear-logarithmic	$T(n) \sim n \log n$
Polynomial	$T(n) \sim n^c$
Exponential	$T(n) \sim c^n$
Factorial	$T(n) \sim n!$

- Note: the base of the logarithm is not important (usually considered 2)
- Great comparison chart & tables here: https://www.bigocheatsheet.com/

Execution time vs. complexity

- Generally speaking, anything below polynomial complexity gives reasonable execution times – polynomial is “doable if must”, but exponential is useless
  - Table values calculated for a CPU of 1 MHz



n \ T(n)	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
10	3 µs	10 µs	30 µs	100 µs	1 ms	1 ms
100	7 µs	100 µs	700 µs	10 ms	1 s	410^{22} a
1000	10 µs	1 ms	10 ms	1 s	17 min	10^{226} a
10^4	13 µs	10 ms	130 ms	100 s	12 d	
10^5	17 µs	100 ms	1,7 s	2,8 h	32 a	
10^6	20 µs	1 s	20 s	12 d	310^4 a	
10^7	23 µs	10 s	4 min	3,2 a	310^7 a	
10^8	27 µs	100 s	44 min	317 a	310^{10} a	
10^9	30 µs	17 min	8,3 h	310^4 a	310^{13} a	

Example: Search algorithms

- Suppose we want to search for an item in a list – say, a student from the university student record (using either a name – or student number, which eliminates the possibility of namesakes)
- Our university has roughly 10 000 students, but there are probably another 10 000 old students in the records, so let’s say the database has 20 000 items
- If we’re searching for a certain student number from the records, the size of our problem is n = 20 000
- Let’s compare two search algorithms

Student number	Student name	…
0024022	Smith, Chad	
0025035	Frusciante, John	
0025594	Kiedis, Anthony	

Example: Search algorithms – sequential search

- Simplest search algorithm is called *sequential search*:
  1) Start from the beginning of the list
  2) Compare the value to the one we’re searching; if it’s the desired one, search is done
  3) If it’s not, move to the next item on the list and repeat step 2
- Very surefire way, but doesn’t seem efficient
- For sequential search, complexities are the following:
  - Best-case scenario: $T(n) \sim 1$ (desired item was the first one on the list, what a luck!)
  - Worst-case scenario: $T(n) \sim n$ (desired item was the last one, or not on the list at all)
  - Average-case scenario: $T(n) = \frac{1}{2}n \sim n$* (statistical odds)
- Conclusion: sequential search is $\Theta(n)$ and $O(n)$

*Warning: Don’t be fooled by this example - finding the average-case complexity is often much harder and requires advanced statistical calculations.

Example: Search algorithms – binary search

- Another, more powerful algorithm is called *binary search*:
  1) Initiation: potential range = the entire list
  2) Examine the middle entry of potential range. If the item is the desired one, search is done
  3) If the middle entry is greater than the desired item, reduce the potential range to entries on the left side of the middle entry. If it's less, reduce the potential range to entries on the right.
  4) Go back to step 2
- This algorithm requires that the list is already ordered! (Now we can assume it is.)
- For binary search, complexities are the following:
  - Best-case scenario: $T(n) \sim 1$ (desired item was the middle one on the list, what a luck!)
  - Worst-case scenario: $T(n) \sim \log n$ (potential range evolution: 20 000 → 10 000 → 5000 → 2500 → 1250 → 625 → etc, so after x steps our potential range will be reduced to 1)
- Conclusion: sequential search is $O(\log n)$ (and also $\Theta(\log n)$*)

$$\frac{n}{2^x} = 1$$

$$n = 2^x$$

$$x = \log n$$

*Calculation of this is skipped.

Example: Search algorithms - comparison

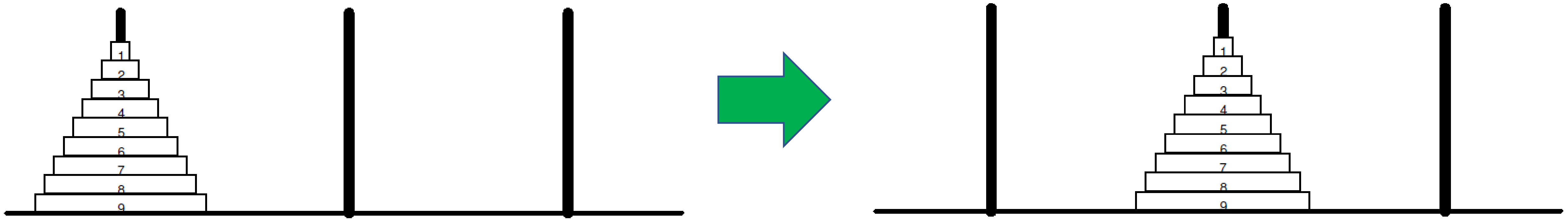
- If our computer is able to perform one comparison in 5 ms, the worst-case search times for algorithms would be:
  - Sequential search – $20\,000 \cdot 5\ ms = 100\,000\ ms = 100\ s = 1$ minute 40 seconds
  - Binary search – $\log(20\,000) \cdot 5\ ms \approx 15 \cdot 5\ ms = 75\ ms\ = 0.075$ seconds
- If you’re a secretary, the waiting time of the sequential search is painfully long
- The binary search, on the other hand, feels like an instant response
- The effect is even greater if we extend this idea to larger problem size – consider, for example, finding a person by social security number in China (n = 1.4 billion):
  - Sequential search – $1\,400\,000\,000 \cdot 5\ ms = 7\,000\,000\ s \approx 81$ days(!!!)
  - Binary search – $\log(1\,400\,000\,000) \cdot 5\ ms \approx 31 \cdot 5\ ms = 155\ ms = 0.155$ seconds – basically, still an instant response!

Example: Towers of Hanoi

- This problem is based on an old legend of a Hindu temple where, according to the legend, were three posts, and in one of them 64 disks stacked from largest to smallest in diameter (two other posts are initially empty)
- The holy mission of the priests was to move all disks from the original post to another post – but with following limitations:
    - Only one disk can be moved at a time.
    - After a move, all disks have to be on a post – they can’t be “moved to the side” to wait
    - At all times, disks have to be in size order – so, from largest to smallest
- The solution to the problem consists of surprisingly many moves
- Let’s perform an illustration for the algorithm using 9 disks (in order to get neater pictures):

Example: Towers of Hanoi

- Pictured: initial state and the desired state



- Let’s break the solution down to smaller parts in a recursive manner:
  - First we have to move all disks except the bottom one to the 3rd post (right)
  - Then we move the bottom disk to the goal post
  - Then we move all disks on the 3rd post to the goal post

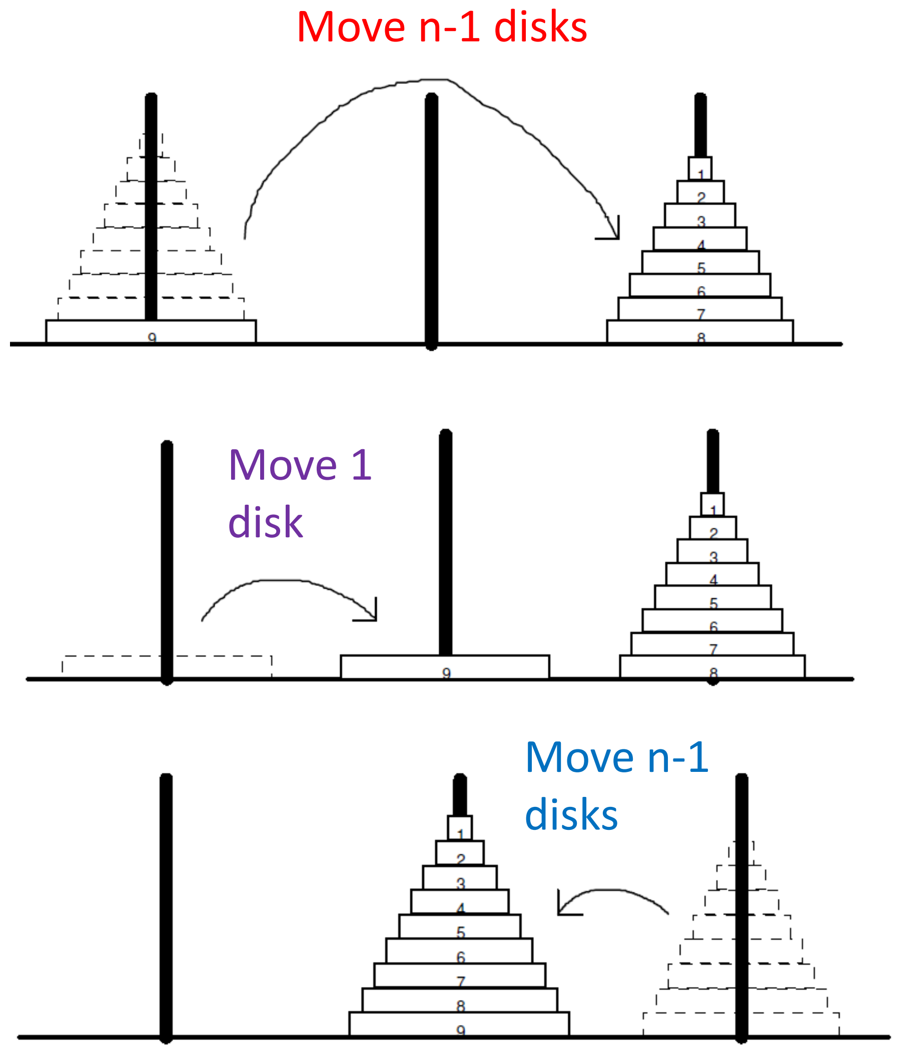
Example: Towers of Hanoi

- Illustration of these 3 steps:
- The complexity was originally T(n)
- Now the complexity can be expressed as

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

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

- This is a recursive algorithm. Let’s continue!



Example: Towers of Hanoi

- Continuing the recursion, we get
  - $T(n) = 2T(n-1) + 1 = 2(2T(n-2) + 1) + 1 = 4T(n-2) + 2 + 1$ (recursion #2)
  - $T(n) = 4(2T(n-3) + 1) + 2 + 1 = 8T(n-3) + 4 + 2 + 1$ (recursion #3)
  - $T(n) = 8(2T(n-4) + 1) + 4 + 2 + 1 = 16T(n-4) + 8 + 4 + 2 + 1$ (recursion #4)
- Let's express the terms in powers of two:
  - $T(n) = 2^4 T(n-4) + 2^3 + 2^2 + 2^1 + 2^0$ (recursion #4)
- I think we can see a pattern in here! After n-1 recursions we have simplified this to a problem that has only one disk. Let's link the powers to the n:
  - $T(n) = 2^{n-1} T(1) + 2^{n-2} + 2^{n-3} + \cdots + 2^1 + 2^0$ (recursion #n-1)
- What is the complexity of T(1)? Naturally it's 1, because moving 1 disk = 1 move
  - $T(n) = 2^{n-1} + 2^{n-2} + 2^{n-3} + \cdots + 2^1 + 2^0$

Example: Towers of Hanoi

- Now, our complexity function is $T(n) = 2^{n-1} + 2^{n-2} + 2^{n-3} + \cdots + 2^1 + 2^0$
- Number of needed elementary operations could be calculated, but how about the classification? A simplified expression would be nice
- Discovery: this T(n) is a geometric series, where $a_1 = 2^0$, $q = 2$ and $n = n$
- Sum of a geometric series-formula:

$$\boldsymbol{S_n = a_1 \frac{1-q^n}{1-q}} \quad\Rightarrow\quad T(n) = 2^0 \cdot \frac{1-2^n}{1-2} = 1 \cdot \frac{1-2^n}{-1} = \boldsymbol{2^n - 1}$$

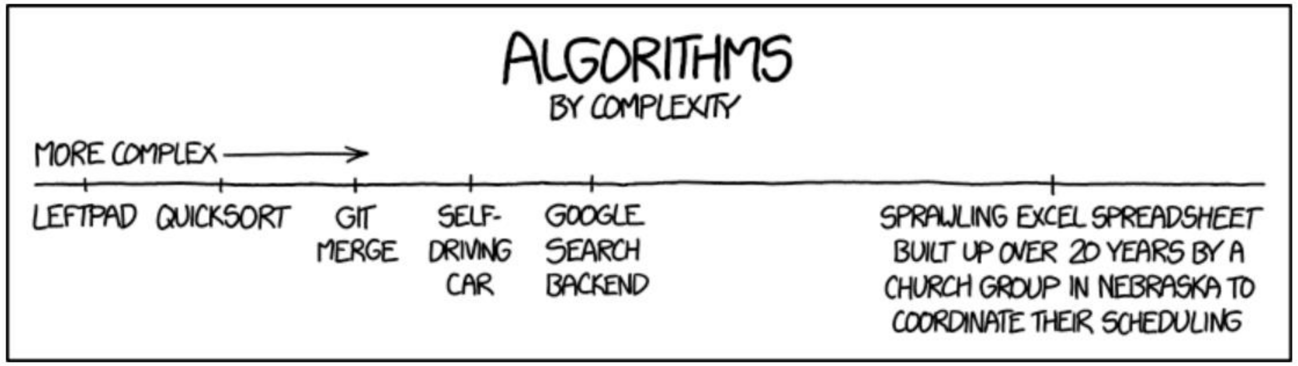
- So, the problem is exponential in complexity: $O(2^n)$
- Number of moves for 64 disk-case: $2^{64} - 1 \approx 1.8447 \cdot 10^{19}$
  - A good animation (3…8 disks) here: https://www.mathsisfun.com/games/towerofhanoi.html

Feasibility of problems

- As we noticed, the complexity of a problem has a huge impact on its computability
- For this reason, problems that can be calculated in polynomial time (i.e. there exists an algorithm that is $O(n^c)$, at maximum) are considered *feasible problems*
- Contrariwise, problems which are exponential (or greater) in complexity are considered *infeasible problems*
- Infeasibility can be a result of two things (or both):
  - Limited number of options, but very high number of moves
  - Limited number of moves, but very high number of options
- Examples of infeasible problems:
  - Towers of Hanoi, traveling salesman problem, playing chess…
  - Scheduling of timetables

How to solve infeasible problems?

- It is not possible to produce complete solutions for infeasible problems
- This kind of problems can still be considered, if we compromise on perfectionism:
  - Limit analysis to special cases of the problem (that can be solved quickly)
  - Construct an algorithm that works quickly for average/most common inputs
  - Approximative algorithms (find a solution that's close)
  - Probabilistic algorithms (find *almost* always a correct solution)
  - Heuristic algoritms (user guides the decision process and discards options which are ”bad” – in many cases, humans can detect these much earlier than a computer!)

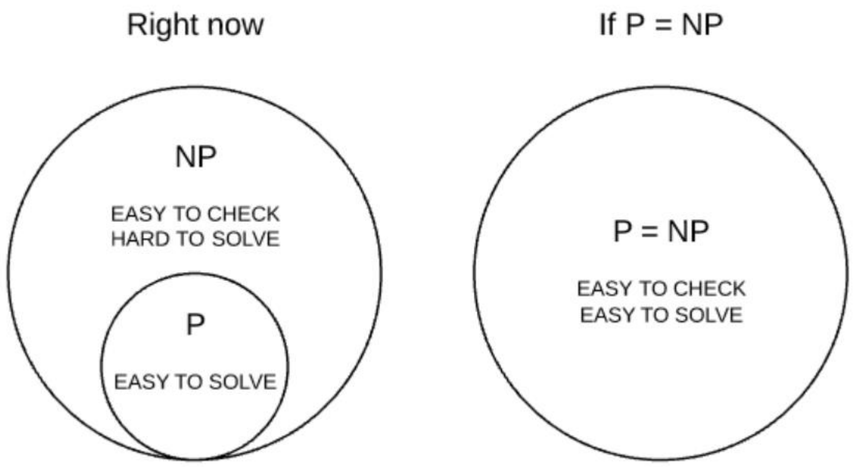

ALGORITHMS
BY COMPLEXITY
MORE COMPLEX
LEFTPAD
QUICKSORT
GIT MERGE
SELF-DRIVING CAR
GOOGLE SEARCH BACKEND
SPRAWLING EXCEL SPREADSHEET BUILT UP OVER 20 YEARS BY A CHURCH GROUP IN NEBRASKA TO COORDINATE THEIR SCHEDULING

P vs NP

- Now when we're familiar with the concept of complexity, let's use another grouping to classify problems
- P = set of problems that can be deterministically solved in polynomial time (so, feasible problems)
- NP = set of problems whose solutions can be verified in polynomial time, but can only be solved in non-deterministic fashion (if at all)
  - For example, finding prime factors of a large number: really hard to find, but when a solution is found, it can be checked by simple multiplication
  - Another example: finding solutions to $A^3 + B^3 + C^3 = 3$ (A, B, C are integers)
- *NP-hard* = set of problems that can't be solved nor checked in polynomial time
  - Many optimization problems (no way to check whether an "optimal" solution is actually optimal or not)

NP-complete problems

- *NP-complete* = subset of NP; set of problems to each which any other NP-problem can be reduced to in polynomial time
- Discovery of a polynomial-time solution for even one NP-complete problem would mean that all other NP-complete problems can also be solved in polynomial time
- This would mean that P = NP
  - This would be a revolutionary discovery – providing an answer for the question ”Is P=NP?” will be rewarded by $1 million from Clay Mathematics Institute
- Current understanding is that $P \subset NP$
  - Hasn’t been proven yet!



Correctness

- Can an algorithm be proven to work correctly in all cases?
    - Tough job, but not impossible
- In an ideal case, the *correctness* of an algorithm can be proven
- There are two types of correctness
    - Functional correctness: the algorithm produces the desired result for each input
    - Total correctness: on top of functional correctness, the algorithm also terminates in finite time
- As the name implies, total correctness is harder to prove
    - For example, if the algorithm searches for a minimum value for some function and this minimum value is achieved by multiple variable values → will the algorithm terminate, or will it alternate between these options indefinitely?

Methods of correctness proof

- A list of methods that are possible to use:
  - Induction proof
  - Use of invariants (= arguments that remain unchanged during the execution), which are proven by induction
  - Proof by consecutive arguments
  - Curry-Howard correspondence (aka Curry-Howard isomorphism)
  - Hoare logic
- Automatization of proofs requires a separate logical system
  - Prolog is a programming language specifically meant for automatic theorem proving
- These methods above won’t be considered in detail on this course
  - You may encounter some of these in later courses, though

“Proof” by testing

- In practice, correctness can be proven only for small programs or program modules; proving the correctness of large programs (such as operating systems) is practically impossible
- Because some kind of verification that the program works correctly is needed, a practical way to approach this is *testing*
- Testing can detect at least the following types of problems:
  - Definition errors (index mistakes, divide by zero-situations, …)
  - Usage errors (user thinks he/she gives the input correctly, but the program interprets it in another way → improve documentation or improve UI/syntax)
- Something to remember: also testers can make mistakes!

Summary

- Problems can be classified based on their computability
- A single problem can be solved by (theoretically indefinitely) many algorithms
  - There are major differences in efficiency – especially as the size of the problem increases
  - (Asymptotic) time complexity is the key
- Computability of a problem is largely defined by the complexity of the best algorithm that solves the problem
- Problems that can be solved in polynomial time are considered feasible
- It is hard to prove completely the correctness of an algorithm – let alone the correctness of a program
- Especially with larger programs we need to rely on extensive testing

Thank you for listening!