UNIVERSITY
OF SKÖVDE

Computation: Complexity Theory

Göran Falkman
School of Informatics
University of Skövde

goran.falkman@his.se

Lecture Overview

UNIVERSITY OF SKÖVDE

- Complexity theory
  - Easy problems (sort a million items in a few seconds)
  - Hard problems (schedule a thousand classes in a hundred years)
  - **What makes some problems hard and others easy** (computationally) and **how do we make hard problems easier**?
  - Complexity theory addresses these questions
- Analysis of the complexity of algorithms
  - **Measuring and analyzing time complexity** – in theory and in practice
- Introduction to computability theory
  - In the first half of the 20th century, mathematicians such as Kurt Gödel, Alan Turing, and Alonzo Church discovered that **certain fundamental problems cannot be solved by computers – efficiently or at all**
  - Computability theory: classify problems as solvable and not solvable
  - P, NP, NP-Hard and NP-Complete classes of algorithms

Motivation

- Analysis of the complexity of algorithms:
  - How to compute, or estimate, the time (and space) complexity of algorithms?
  - **What effects the complexity of computer programs?**
- Why study complexity theory?
  - Some problems are intrinsically easy and some are intrinsically hard
  - Which problems can efficiently be solved by a computer?
  - **Decide on pros and cons of different solutions**
- Why study computational theory?
  - Some problems are solvable and some are not solvable
  - **What can and cannot be computed – in theory and in practice – with a computer as we know it?**
- Practice vs. theory:
  - Worst-case vs. average case vs. asymptotic (long-run) complexity

What Effects Time Complexity?

- Performance (speed) of the hardware
- Quality of the compiler/interpreter (the code it generates)
- Skill of the programmer
- Size of the problem instance and the characteristics of the input
- Choice of algorithm
- The inherent complexity of the problem

Complexity Analysis

UNIVERSITY OF SKÖVDE

- How to measure the time complexity of programs?
  - **Empirically**: Run the program for different inputs and measure the computation time:
    - » Expensive and problematic
    - » Cannot test every possible input
    - » Can only be performed after implementation
  - **Amortized analysis**:
    - » How does the program perform during the long run of its lifetime?
    - » Even more problematic
  - **Theoretically**: Perform a rigorous mathematical analysis of the underlying algorithm as run on a general computational platform:
    - » Can be done before resources are put into the implementation of the program
    - » Gives an undisputable "truth"

Time Complexity

- Time complexity as **a function of the size of the input to the program**:
  - $T(n)$ = the time it takes to perform an algorithm on input of size n
  - Interested in the case when n is (very) large, i.e., in **asymptotic complexity**
- Running time is not everything:
  - For programs that are seldom used development costs dominates everything else
  - If small inputs dominate, an advanced algorithm may bring unnecessary overhead
  - An efficient algorithm is often very complex, i.e., hard to understand and maintain
  - A time-wise efficient algorithm may require large amounts of memory
  - For numeric algorithms exactness and stability are equally important

How to Analyze Time Complexity?

- Assume the use of a sequential computer (von Neumann architecture) with infinite memory capacity and instant access to input and output streams (no interaction):
  - All atomic statements have a cost of 1 time unit:
    - » Assignment, arithmetic and logical operations, indexing of memory structures, input and output operations etc.
    - » Cost of consecutive instructions is the sum of the individual elements
  - Iterations:
    - » While-loops: (Cost of test · (#iterations + 1)) + (cost of body · #iterations)
    - » For-loops: Cost of initializations + (cost of test · (#iterations + 1)) + (cost of body · #iterations) + (cost of increasing counter · #iterations)
    - » Nestled iterations are analyzed from the innermost loop and out
  - Selections: Cost for test + *max*(cost of the alternatives)
  - Results in **an approximation of the total time taken**, where statements that depend on the input size are decisive – **frequency count in terms of input size**

Time Complexity: Simple Examples

Program 1

```
x := x + 1
```

Frequency = 1

Program 2

```
FOR i := 1 to n
DO
   x := x + 1
END
```

Frequency = n

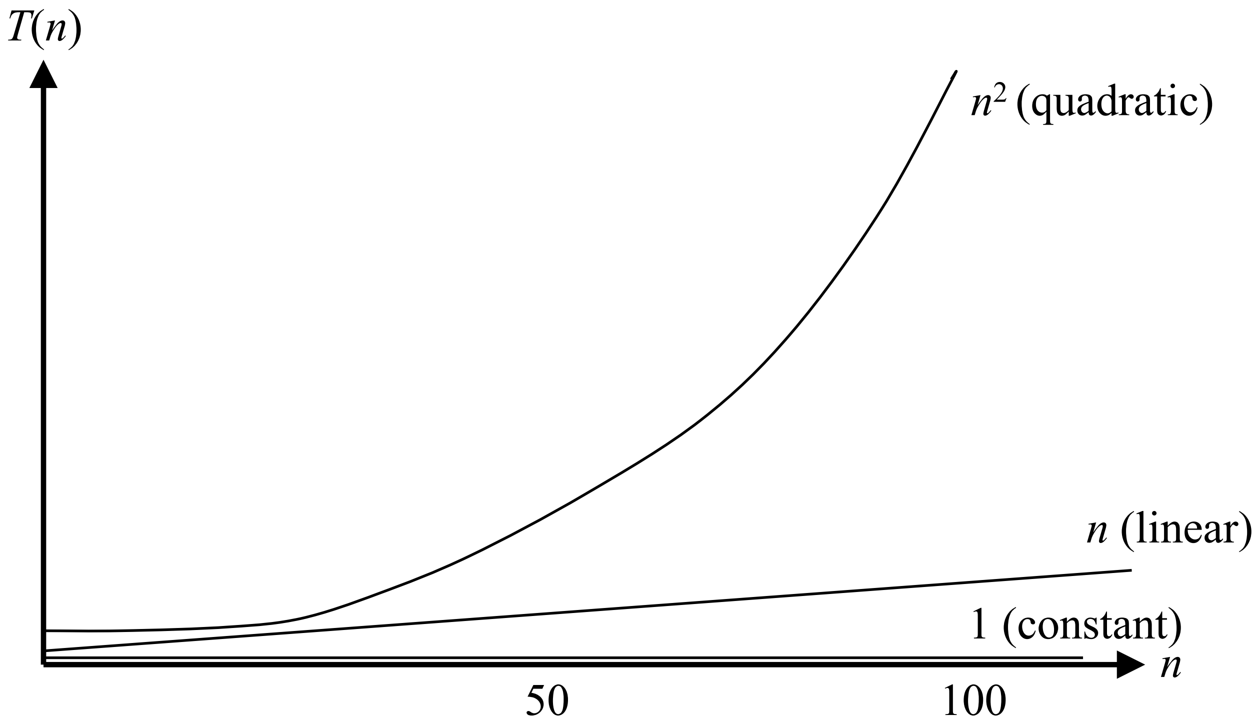
Program 3

```
FOR i := 1 to n
DO
   FOR j := 1 to n
   DO
      x := x + 1
   END
END
```

Frequency = n^2

Orders of Magnitude

- 1, n, and n^2 are said to be different and increasing **orders of magnitude** (e.g., if n = 10 then we get 1, 10 and 100)
- We are interested in determining the order of magnitude of the time complexity of an algorithm. Why?



Time Complexity: Fibonacci Sequence

UNIVERSITY OF SKÖVDE

- Let’s look at an algorithm to print the n^{th} term of the Fibonacci sequence:

  0 1 1 2 3 5 8 13 21 34 …

  as given by

  $t_0 = 0$

  $t_1 = 1$

  $t_n = t_{n-1} + t_{n-2}$

```
                                            n > 1
procedure fibonacci {print nth term}        0
    read(n)                                 1
    if n<0                                  1
        then print(error)                   0
    else if n=0                             1
        then print(0)                       0
    else if n=1                             1
        then print(1)                       0
        else                                1
            fnm2 := 0;                      1
            fnm1 := 1;                      1
            FOR i := 2 to n DO              n
                fn := fnm1 + fnm2;          n-1
                fnm2 := fnm1;               n-1
                fnm1 := fn                  n-1
            end                             n-1
            print(fn);                      1
```

Big-O Notation

- The cases where $n < 0$, $n = 0$ and $n = 1$ are not particularly instructive or interesting. In the case where $n > 1$, we have the total statement frequency of:

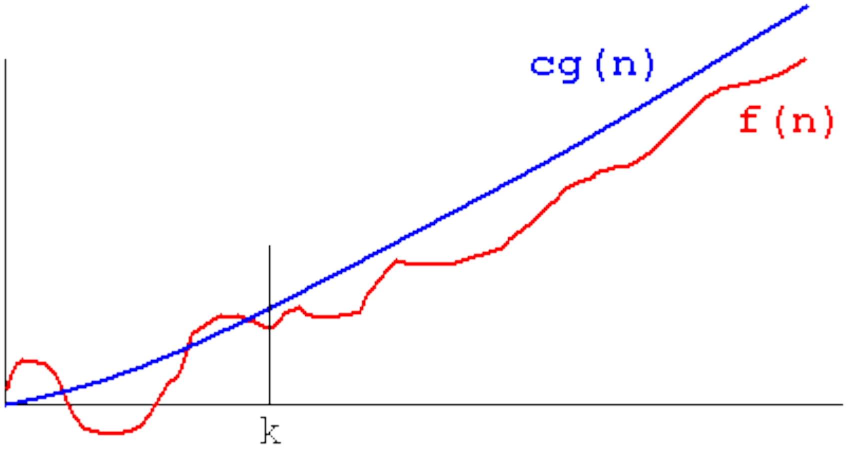
  $$8 + n + 4(n-1) = 5n + 4$$

- We write this as $5n + 4 = O(n)$, thus ignoring the constants
- More formally, $f(n) = O(g(n))$ – read as "f of n is big-O of g of n" – where $g(n)$ is an asymptotic upper bound for $f(n)$:

  $f(n) = O(g(n))$ if there exist two constants c and k such that $f(n) \leq c \cdot g(n)$ for all $n \geq k$

"$f(n)$ never gets worse than $g(n)$ (for large enough n)"

Big-O Notation



Suppose $f(n) = 2n^2 + 4n + 10$.

Then $f(n) = O(g(n))$ for $g(n) = n^2$, $c = 16$ and $k = 1$:

$$f(n) = 2n^2 + 4n + 10$$

$$f(n) \leq 2n^2 + 4n^2 + 10n^2$$

for $n \geq 1$

$$f(n) \leq 16n^2$$

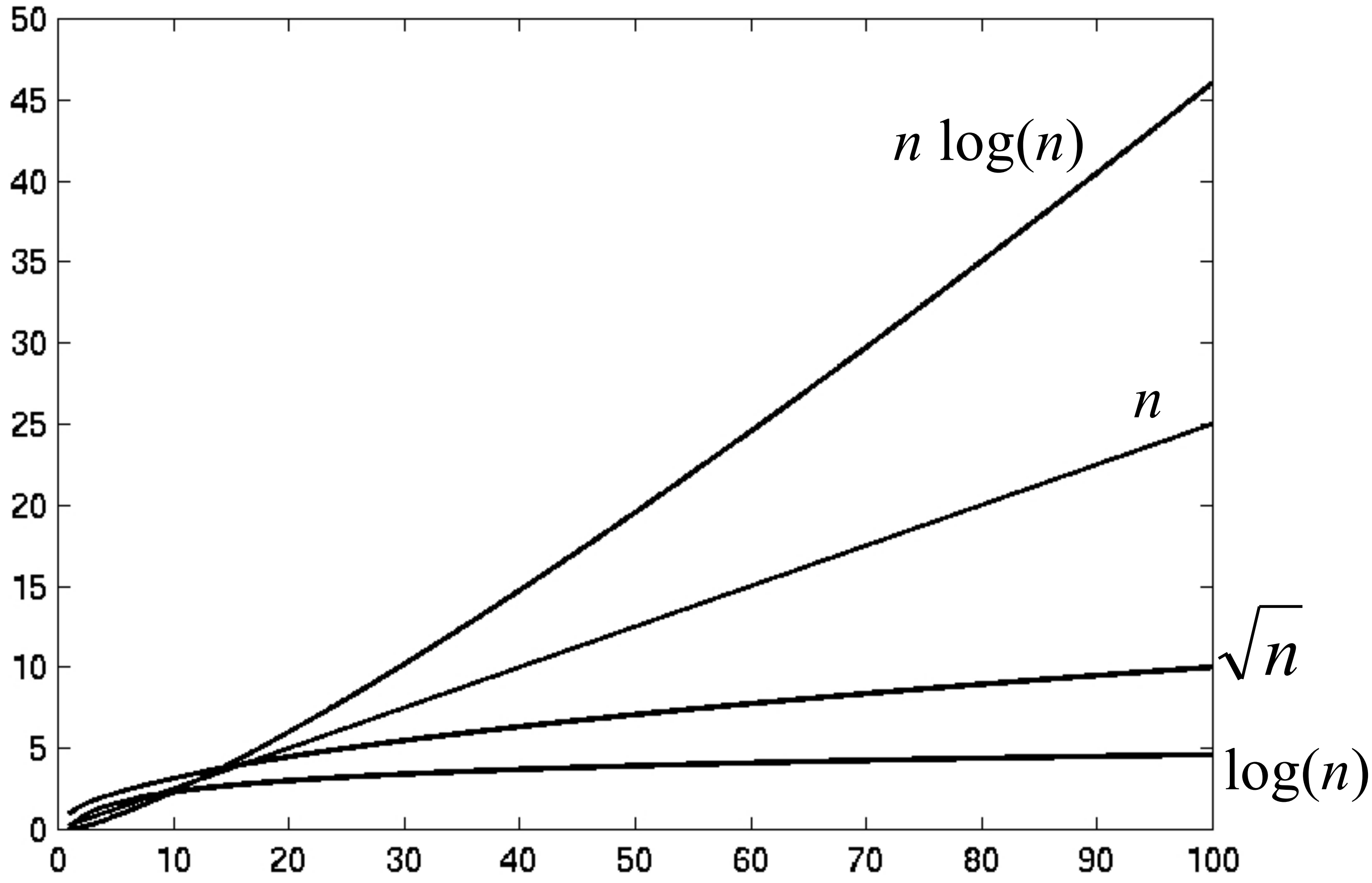
$$f(n) \leq 16g(n)$$

Usually one seeks **as tight an upper bound as possible**

Arithmetic of Big-O Notation

- If $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$ then $T_1(n) + T_2(n) = O(max(f(n), g(n))$
- If $f(n) \le g(n)$ then $O(f(n) + g(n)) = O(g(n))$
- If $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$ then $T_1(n) \cdot T_2(n) = O(f(n) \cdot g(n))$
- Examples:
  - $f_1(n) = 10n + 25n^2 \Rightarrow O(f_1(n)) = O(max(10n, 25n^2)) \Rightarrow$ $O(f_1(n)) = O(max(O(10) \cdot O(n), O(25) \cdot O(n^2))) \Rightarrow O(f_1(n)) = O(max(1n, 1n^2))$ $\Rightarrow O(f_1(n)) = O(max(n, n^2)) \Rightarrow O(f_1(n)) = O(n^2)$
  - $f_2(n) = 20(n \log n) + 5n \Rightarrow \ldots \Rightarrow O(f_2(n)) = O(n \log n)$
  - $f_3(n) = 12(n \log n) + 0.05n^2 \Rightarrow \ldots \Rightarrow O(f_3(n)) = O(n^2)$
  - $f_4(n) = n^{1/2} + 3(n \log n) \Rightarrow \ldots \Rightarrow O(f_4(n)) = O(n \log n)$

Orders of Magnitude Revisited

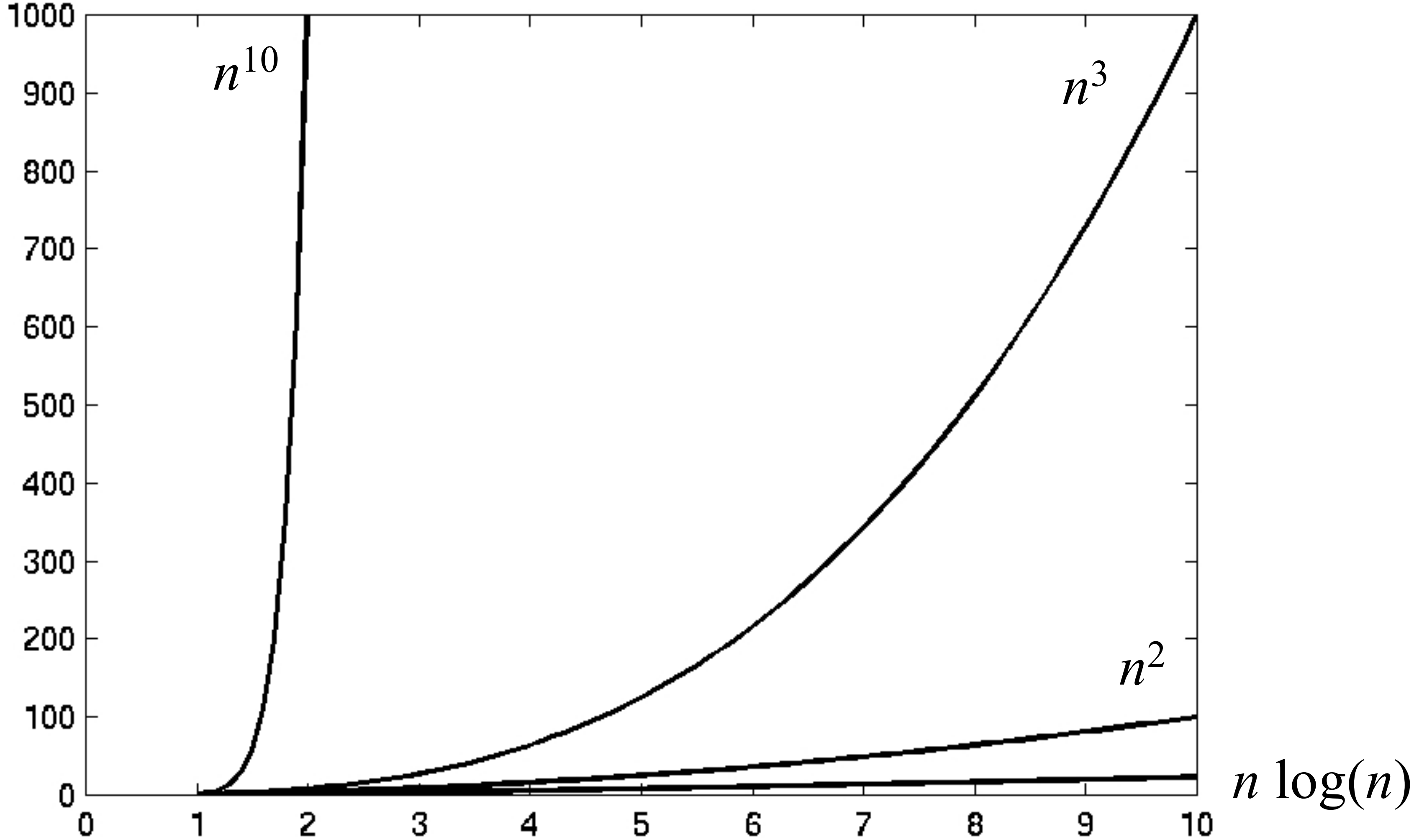


Orders of Magnitude Revisited (2)

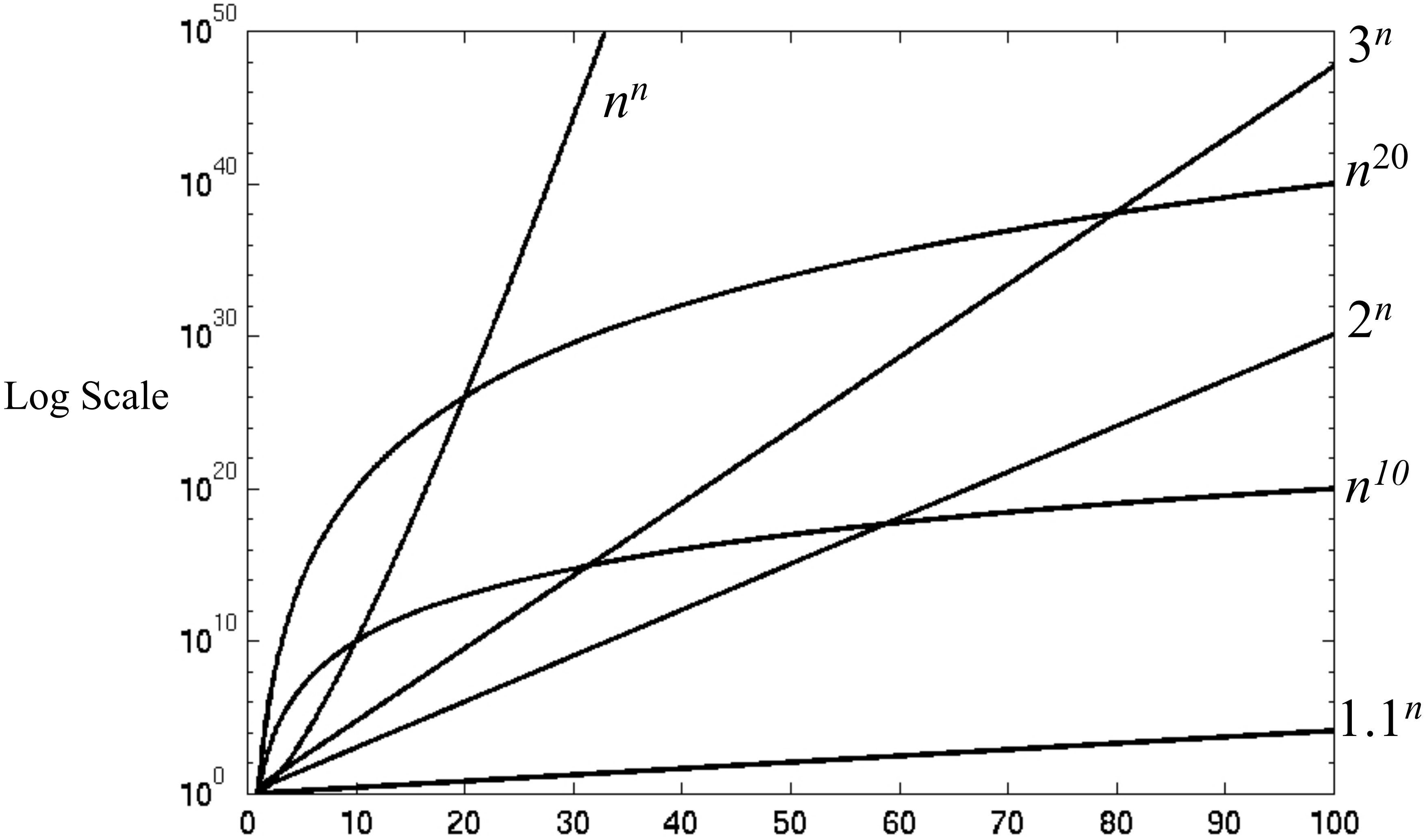
UNIVERSITY OF SKÖVDE

$O(1)$	**Constant** (computing time)
$O(\log n)$	**Logarithmic** (computing time) is faster than $O(n)$ for sufficiently large n
$O(n)$	**Linear** (computing time)
$O(n \log n)$	**"n log n"** is faster than $O(n^2)$ for sufficiently large n
$O(n^2)$	**Quadratic** (computing time)
$O(n^3)$	**Cubic** (computing time)
$O(2^n)$	**Exponential** (computing time)

Orders of Magnitude Revisited (3)



Orders of Magnitude Revisited (4)



Orders of Magnitude Revisited (5)

UNIVERSITY OF SKÖVDE

- The effect on running time $T(n)$ if n is doubled in size:

Constant	no increase: $T(2n) = T(n)$
$\log n$	small increase: $T(2n) > T(n)$
n (*linear*)	doubled running time: $T(2n) = 2T(n)$
$n \log n$	little more than doubled running time: $T(2n) > 2T(n)$
n^2	increase with a factor 4: $T(2n) = 4T(n)$
n^3	increase with a factor 8: $T(2n) = 8T(n)$
2^n	quadratic increase: $T(2n) = T(n)^2$

Orders of Magnitude Revisited (6)

UNIVERSITY OF SKÖVDE

- If we had a computer a 1,000, 1,000,000 or 1,000,000,000 times faster …

Operations per second	$n = 10^6$			$n = 10^9$		
	n	$n \log n$	n^2	n	$n \log n$	n^2
10^6	seconds	seconds	weeks	hours	hours	“never”
10^9	instantly	instantly	hours	seconds	seconds	decades
10^{12}	instantly	instantly	seconds	instantly	instantly	weeks
10^{15}	instantly	instantly	instantly	instantly	instantly	minutes

Complexity of Recursive Algorithms

$n!$ (factorial) $= 1 \cdot 2 \cdot 3 \cdot \ldots \cdot \mathrm{n}$

$$fac(0) = fac(1) = 1$$
$$fac(n) = n \cdot fac(n-1)$$

```
                                                    n > 1
int factorial(int n){
   int factorial_value;                             1
   factorial_value = 0;                             1
   if (n <= 1)                                      1
      factorial_value = 1;                          0
   else
      factorial_value = n * factorial(n-1);         2 + T(n-1)
   return (factorial_value);                        1
}
```

Complexity of Recursive Algorithms (2)

- In general, this is more difficult:
  - Identify and solve the **recurrence relation** implicit in the recursion

$$
\begin{aligned}
T(n) &= 6 + T(n-1) \Rightarrow T(n) = c + T(n-1) \\
T(n-1) &= c + T(n-2) \Rightarrow T(n) = c + c + T(n-2) = 2c + T(n-2) \\
T(n-2) &= c + T(n-3) \Rightarrow T(n) = 2c + c + T(n-3) = 3c + T(n-3) \\
&\dots \\
T(n) &= ic + T(n-i\ )
\end{aligned}
$$

Finally, when $i = n-1$ the unfolding stops and we get:

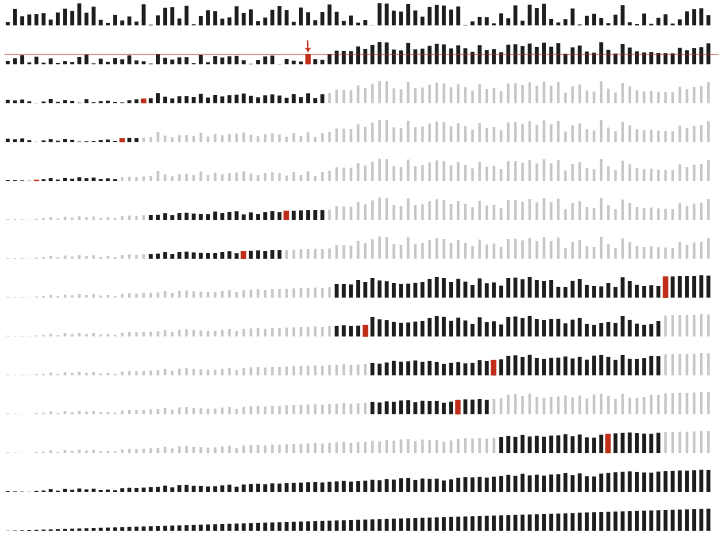
$$T(n) = (n-1)c + T(n-(n-1)) = (n-1)c + T(1) = (n-1)c + d$$

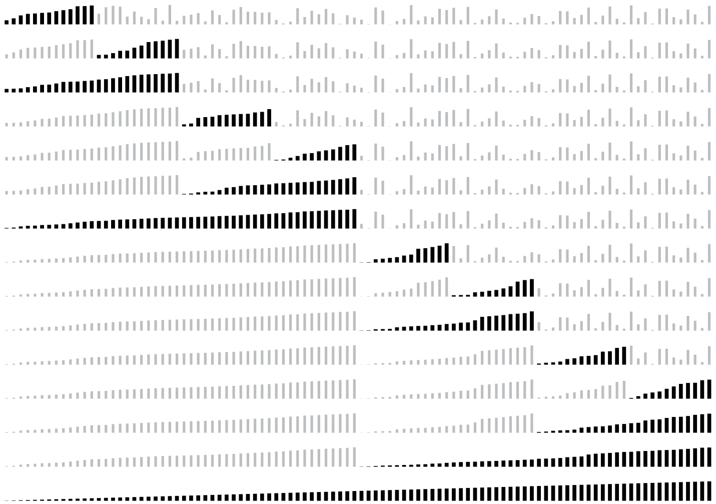
Hence, $T(n) = O(n)$

Worst-, Average- and Best-Case Complexity

- So far, we have looked only at worst-case time complexity, i.e., we have developed an upper-bound on running time

- However, there are times when we are more interested in the **average-case complexity** or **best-case complexity** (especially if they differ significantly):
  - One of the fastest sorting algorithms, Quicksort, has $T(n) = O(n^2)$ worst-case complexity (for inversely sorted data), but $T(n) = O(n \log^2 n)$ average-case complexity (for randomly ordered data). Also the best-case complexity is $T(n) = O(n \log^2 n)$
  - For another very fast sorting algorithm, mergesort, both the worst-, average- and best-case complexity is $T(n) = O(n \log^2 n)$, i.e., mergesort is **more stable**





Towers of Hanoi

- The goal of this puzzle is to transfer all n disks from peg A to peg C using auxiliary peg B according to the following rules:
  - you can only move one disk at a time
  - you can never place larger disk above a smaller one
- Recursive solution:
  - transfer $n-1$ disks from A to B
  - move largest disk from A to C
  - transfer $n-1$ disks from B to C
- Recurrence relation:

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

  $T(1) = 1$

Complexity of the Towers of Hanoi

- Solution by unfolding, which stops when $i = n-1$:

$$\begin{aligned} T(n) &= 2(2T(n-2) + 1) + 1 = 4T(n-2) + 2 + 1 \\ &= 4(2T(n-3) + 1) + 2 + 1 = 8T(n-3) + 4 + 2 + 1 = \ldots \\ &= 2^i \cdot T(n-\mathrm{i}) + 2^{i-1} + 2^{i-2} + \ldots + 2^1 + 2^0 \\ &= 2^{n-1} + 2^{n-2} + 2^{n-3} + \ldots + 2^1 + 2^0 \end{aligned}$$

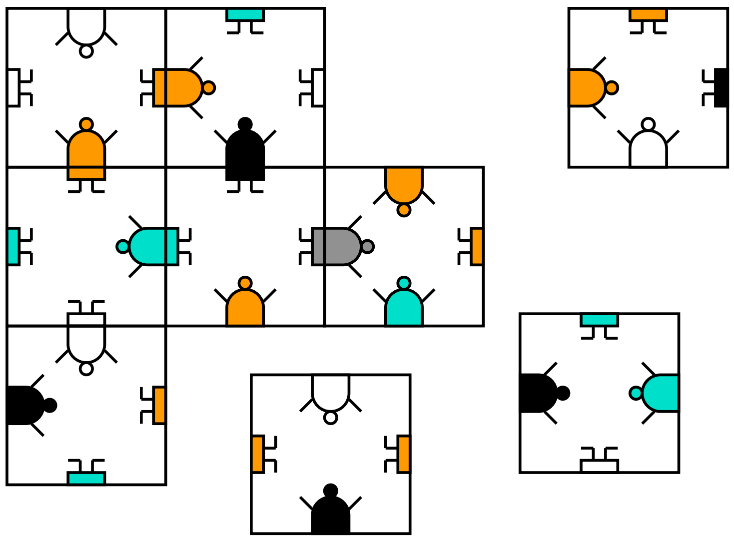
  This is a geometric sum, so we have $T(n) = 2^n-1 = O(2^n)$

- The running time of this algorithm is exponential (k^n) rather than polynomial (n^k)
- Good or bad news?
  - The Tibetan monks were confronted with a tower of 64 golden rings
  - Assuming one could move 1 million rings per second, it would take the monks half a million years to complete the process ...

Monkey Puzzle

- Are such long running times linked to the size of the solution of an algorithm?
- No! To show that, we in the following consider only True/False or Yes/No problems – **decision problems**

- Nine cards with imprinted “monkey halves” with fixed orientations
- Does any 3 × 3 arrangement of matching halves exist?
- A brute-force algorithm to verify whether a solution exists is $O(9!)$



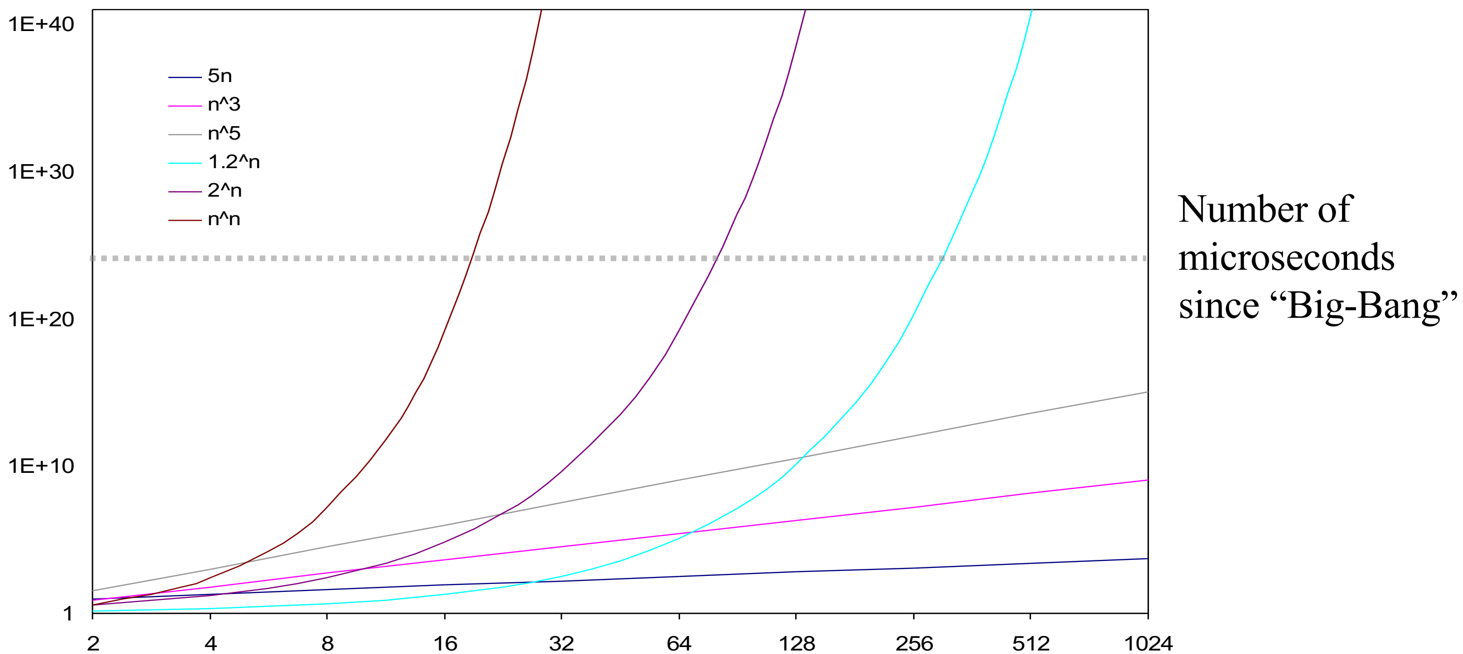
Monkey Puzzle

- Brute force solution: **Go through all possible arrangements**. Assuming the number of cards is 25:
  - Pick a card and place it – there are 25 possibilities for the first placement
  - Pick the next card and place it – there are 24 possibilities
  - Pick the next card, there are 23 possibilities ...
- There are $25 \cdot 24 \cdot 23 \cdot 22 \cdot \ldots \cdot 2 \cdot 1$ possible arrangements, i.e., factorial 25 possible arrangements (25!)
- 25! contains 26 digits, and with 1,000,000 arrangements per second it would take 490 billion years to solve the puzzle
- A smarter algorithm can improve on this by discarding partial arrangements, but would still need thousands of years
- Is there an easier way to find solutions? Perhaps, but nobody has found one, yet …

Reasonable vs. Unreasonable Algorithms

UNIVERSITY
OF SKÖVDE

- The order of complexity of the brute force algorithm for the Monkey Puzzle is $O(n!)$ and $n!$ grows at a rate which is **orders of magnitude larger than polynomial functions**
- Other functions exist that grow even faster, e.g., n^n (super-exponential)
- Even functions like 2^n have unacceptable sizes even for modest values of n



Reasonable vs. Unreasonable Algorithms

UNIVERSITY OF SKÖVDE

	function/ n	10	20	50	100	300
Polynomial	n^2	1/10,000 second	1/2,500 second	1/400 second	1/100 second	9/100 second
	n^5	1/10 second	3.2 seconds	5.2 minutes	2.8 hours	28.1 days
Exponential	2^n	1/1000 second	1 second	35.7 years	400 trillion centuries	a 75 digit-number of centuries
	n^n	2.8 hours	3.3 trillion years	a 70 digit-number of centuries	a 185 digit-number of centuries	a 728 digit-number of centuries

Tractable vs. Intractable Problems

Tractable problems

”Good”, reasonable algorithms are known, i.e., complexity is bound by a polynomial function $T(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$

Intractable problems

Only ”bad”, unreasonable algorithms are known, i.e., complexity is bound by a superpolynomial (exponential) function $T(n) = a^n$

So What!

UNIVERSITY OF SKÖVDE

- Computers become faster every day!
  - As we've seen, for these algorithms it doesn't matter if we get a computer a billion, or trillion, times faster
  - **The number of computed operations per second is insignificant** – just a constant – compared to expected total running time
- This only applies to "toy examples" as the Monkey puzzle, which are of no concern!
  - The Monkey puzzle is just an illustrative example of **a general class of important optimization problems** that fall into a category called NPC
  - **NPC (NP-Complete)** includes ~1000 fundamentally important problems, e.g.,
    - » Travelling Salesman Problem (TSP)
    - » Graph Coloring Problem
    - » Satisfiability Problem (SAT)
    - » Clique Problem
  - For all these problems, no reasonable algorithm is known, i.e., **they are all intractable problems**

Travelling Salesman Problem (TSP)

- TSP is the problem of a salesman who wants to find, starting from his home town, a shortest possible trip through a given set of customer cities and to return to its home town, visiting exactly once each city

- **Naive solutions are O(n!)**, where n is the number of cities
- **Best known exact solution is $O(n^2 \cdot 2^n)$** (from 1962)
- Heuristic (approximate) solutions have manage to solve the problem for 85,900 cities (from 2006)
- The minimum trip visiting all of Sweden's 24,978 municipalities is 72,500 km long (from 2004)

P and NP Complexity Classes

UNIVERSITY OF SKÖVDE

- P is the set of all **decision problems solvable in polynomial time on a deterministic Turing machine**, i.e., on a computer as we know it
- NP is the set of all **decision problems solvable in polynomial time on a nondeterministic Turing machine**, i.e., on an imaginary "oracle-based" computer**:**
  - For all decision points in an algorithm we can, in constant time, **make a perfect guess** on the best choice (e.g., pick the right Monkey Card or select the optimal next leg in the Traveling salesman's trip)
- Alternatively: NP is the set of all **decision problems for which solutions can be verified through a reasonable (polynomial) algorithm** (the verifier) by providing a proof (certificate)

P and NP Complexity Classes (2)

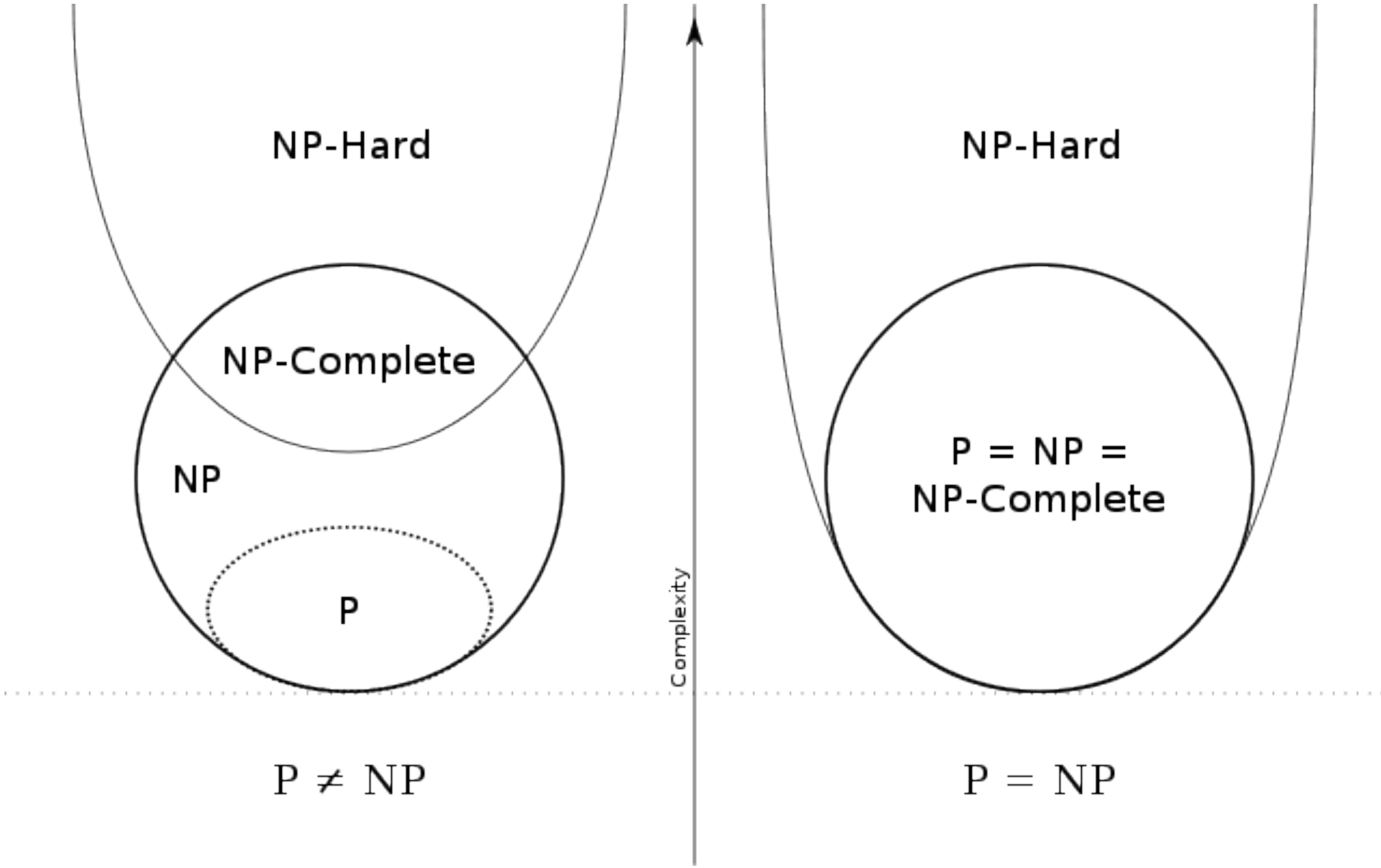
- From before we know that the Monkey Puzzle is not in P. However, given a proposed arrangement of the Monkey Puzzle cards **you can easily (i.e., in polynomial time) check whether the arrangement is a solution** to problem or not, i.e., the Monkey Puzzle is NP:
  - We only need to provide a solution – a certificate
- From before we know that the Traveling Salesman is not in P. However, given a proposed trip **you can not easily (i.e., not in polynomial time) check whether the trip is an optimal solution to the problem or not**, i.e., the Traveling Salesman is not NP:
  - Checking whether the proposed solution is a certificate or not has the same complexity as the original problem

NP-Hard and NP-Completeness

UNIVERSITY OF SKÖVDE

- A NP-Hard (non-deterministic polynomial-time hard) problem is **at least as hard as the hardest problems in NP:**
  - A problem *H* is NP-hard when every problem *L* in NP can be reduced in polynomial time to *H*
- NP-Complete (NPC) problems are NP problems that are NP-hard:
  - The NP-complete class of problems include **the computationally hardest problems known**
- Each NPC problem's fate is tightly coupled to all the others:
  - Finding a polynomial time algorithm for one NPC problem would automatically **yield an a polynomial time algorithm for all NP problems** – Either all NPC problems are tractable or none of them are
  - Proving that one NPC problem has an exponential lower bound would automatically prove that **all other NP-complete problems have exponential lower bounds** – Either all NPC problems are intractable or none of them are

The Big Question



The Big Question (2)

UNIVERSITY OF SKÖVDE

- If P = NP, then
  - There are efficient algorithms for TSP and factoring
  - Cryptography is impossible on conventional machines
  - Modern banking system will collapse
- If not, then
  - Can't hope to write efficient algorithm for TSP
  - But maybe efficient algorithm still exists for testing the primality of a number – i.e., there are some problems that are NP, but not NP-complete
- Probably no, since
  - Thousands of researchers have spent four decades in search of polynomial algorithms for many fundamental NP-complete problems without success
  - **Consensus opinion: P ≠ NP**
- But maybe yes, since
  - **No success in proving P ≠ NP** either

Summary

- Complexity theory:
  - A **polynomial-time algorithm** is one that is bounded from above by some function n^k for some fixed value of k:
    - » **Reasonable algorithm**
  - A **superpolynomial (exponential and super-exponential) time algorithm** is one that is bounded from above by some function k^n for some fixed value of k:
    - » **Unreasonable algorithm**
- Computational theory:
  - **P** – class of problems which admit deterministic polynomial-time algorithms
  - **NP** – class of problems which admit non-deterministic polynomial-time algorithms
  - **NP-Hard** – problems at least as hard as NP problems (every NP problem can be transformed to an NP-Hard problem in polynomial time)
  - **NP-Complete** – NP-problems that are NP-hard

Summary (2)

UNIVERSITY OF SKÖVDE

- So, is NP = P or not?
  - We don't know!
  - The NP = P? problem has been open since it was posed in 1971 and is one of the most difficult unresolved problems in computer science
- It is **not known whether the whole class of NP problems are tractable or intractable**
- But, there exist provably intractable problems
  - Even worse – there exist problems with running times unimaginably worse than exponential
- More bad news: there are **provably noncomputable (undecidable)** problems
  - There are no (and there will never be) algorithms to solve these problems