



Schweizerische Eidgenossenschaft  
Confédération suisse  
Confederazione Svizzera  
Confederaziun svizra



# WARNING



## **THIS PRESENTATION CONTAINS SOME FANCY LOOKING MATH FORMULAS**

The Federal Office of Public Health mandates that anybody displaying such formulas must ensure that those not strongly affectionated to mathematics are made fully aware of their presence prior to appearance on screen.

Anyone experiencing uncontrollable rage, nausea or dizziness should close their eyes and think of pictures of cute kittens.

*The presenter accepts no liability or responsibility for any adverse reaction to the formulas contained in this presentation.*



Universität  
Zürich<sup>UZH</sup>

Institut für Informatik

# Informatics II

## Tutorial Session 4

Wednesday, 16<sup>th</sup> of March 2022

Discussion of Exercise 3,  
Asymptotic Complexity, Running Time Analysis, Invariants

14.00 – 15.45

BIN 0.B.06



## Agenda

- Asymptotic Complexity
  - Task 2 of Exercise 3, Task from Past Exam
- Running Time Analysis
  - Tasks 1 and 3 of Exercise 3
- (Preview on Exercise 4)
- Summary and Wrap-Up



# Asymptotic Complexity, Landau Notation

- Definition and First Illustrated Examples
- Methods for Comparing Asymptotic Complexity of Functions
- Rules for and Notes on Asymptotic Complexity
- Application on C Code Fragments



## Algorithmic Complexity, Landau Notation

- Algorithmic «complexity» is actually not a very good name choice since it has **nothing to do with how complicated an algorithm is**. A very simple algorithm may have a high asymptotic complexity and oftentimes having a low running time complexity needs a very complicated algorithm. A better term may have been «algorithmic **efficiency**» / «**performance**» or something like this.  
(There are other notions of complexity, e.g. cyclomatic complexity / McCabe complexity – not part of this lecture.)
- It's **a way to compare algorithms** with respect to certain criteria (time, memory) while abstracting from details of the technical environment (CPU speed etc.).

## Big O Notation: Formal Definition

Definition:

$$f(n) \in O(g(n))$$

iff  $\exists$  constants  $n_0 > 0$ ,  $c > 0$  such that

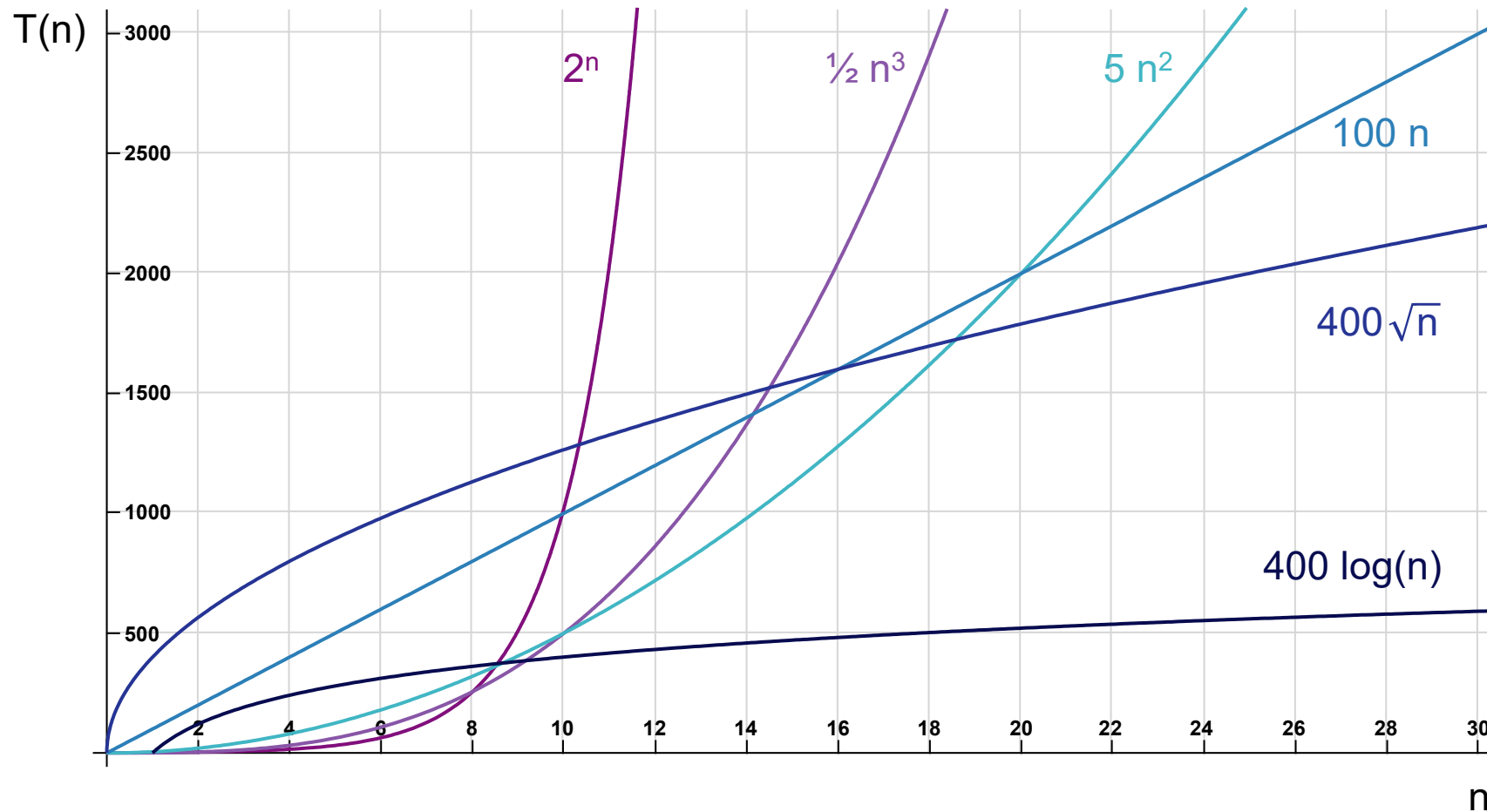
$$f(n) \leq c \cdot g(n) \quad \forall n \geq n_0$$

*From some point on the  $n$ -axis, the values of the functions  $f(n)$  and  $g(n)$  are always constraint / bounded by a constant multiplication factor.*

## Big O Notation: Types of Complexity Measures

Symbol	Symbol name	Meaning	Analogy	Definition	Example
<b>O</b>	Big Omicron	Upper bound	$f \in O(g)$ : " $f \leq g$ "	$\exists n_0 > 0, c > 0: \forall n \geq n_0,$ $f(n) \leq c \cdot g(n)$	$4n + 2 \in O(n^2 + n - 7)$
<b><math>\Omega</math></b>	Big Omega	Lower bound	$f \in \Omega(g)$ : " $f \geq g$ "	$\exists n_0 > 0, c > 0: \forall n \geq n_0,$ $f(n) \geq c \cdot g(n)$	$n^4 - 8 \in \Omega(5n^2 - 2n + 1)$
<b><math>\Theta</math></b>	Theta	Tight bound	$f \in \Theta(g)$ : " $f = g$ "	$\exists n_0 > 0, c_1 > 0, c_2 > 0:$ $\forall n \geq n_0,$ $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$	$9n^2 - 3n \in \Theta(n^2 + 31)$
<b>o</b>	Little Omicron	Strict upper bound	$f \in o(g)$ : " $f < g$ "	$\exists n_0 > 0, \forall c > 0: \forall n \geq n_0,$ $f(n) < c \cdot g(n)$	$6n^2 \in o(2n^3)$
<b><math>\omega</math></b>	Little Omega	Strict lower bound	$f \in \omega(g)$ : " $f > g$ "	$\exists n_0 > 0, \forall c > 0: \forall n \geq n_0,$ $f(n) > c \cdot g(n)$	$7n^3 \in \omega(4n)$

## Asymptotic Complexity Illustrated

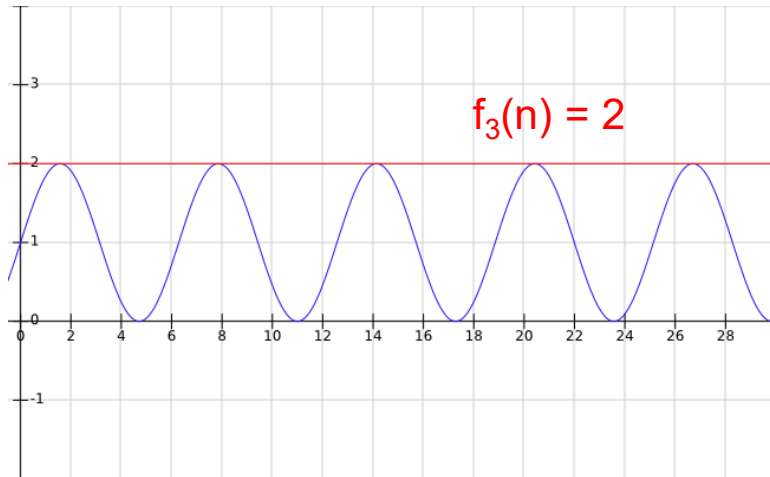




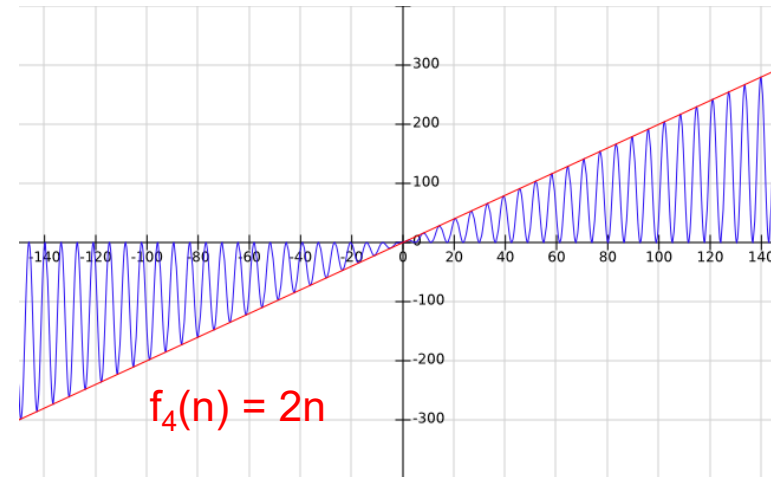
## Asymptotic Complexity Example

Example: What is the asymptotic complexity of the following functions?

$$f_1(n) = 1 + \sin(n) \rightarrow O(1)$$



$$f_2(n) = n \cdot (1 + \sin(n)) \rightarrow O(n)$$



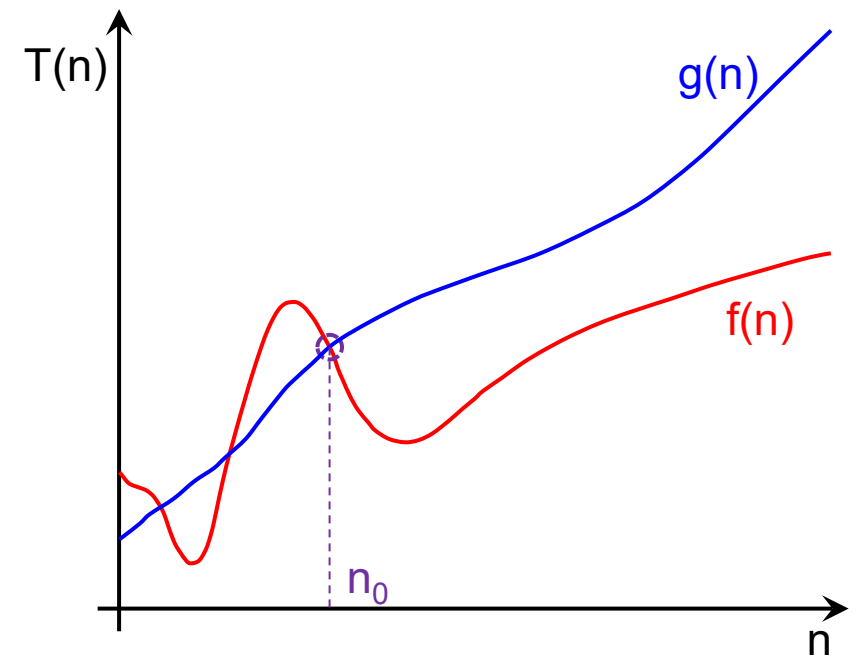
## Asymptotic Complexity Definition: Potential Misconceptions

Remember the definition of the asymptotic upper bound:

$f(n) \in O(g(n))$  iff  $\exists$  constants  $n_0 > 0$ ,  $c > 0$   
such that  $f(n) \leq c \cdot g(n) \quad \forall n \geq n_0$

Which of the following statements are true, which are false?

- a) «If  $f(n) \in O(g(n))$  holds, then all values of  $g(n)$  have to be bigger than the values of  $f(n)$  beyond  $n_0$ .» *false*
- b) «If  $f(n) \in O(g(n))$  holds, there cannot be any intersection of the two functions beyond  $n_0$ .» *false*
- c) «The value of the constant  $c$  from the definition has to be chosen such that  $n_0$  is at an intersection point of the two functions in question.» *false*



## Methods for Finding the Asymptotic Complexity

There are several ways to find the asymptotic complexity of a given function  $f(n)$  or to prove that they are or are not in the same complexity class:

- **Plot / graphical argument / «plug 'n' play»:** plugging large numbers in the functions and hoping for the best  
Is no proof, but can be helpful for intuition; may be misleading.
- **Algebraic transformation** into functions for which the complexity is already known  
Use rules for multiplication and addition of Big O expressions.
- Form **limes (superior) of the fraction of the functions**.
- **Take the logarithm** of both functions which should be compared and compare their logarithms to each other.
- Find (or guess) and **prove  $n_0$  and  $c$  from definition**.

## Hints for Exercises and Exam Tasks on Asymptotic Complexity

- In tasks where you are requested to determine the asymptotic complexity of functions, it is (unless explicitly stated differently) *not necessary to prove the ranking of basic function classes*, e.g. it is considered for granted that  $n^n$  grows faster than  $2^n$  or that  $n^{1/2}$  grows faster than  $\log(n)$ . Thus it is sufficient to *know the ordering of function classes* in terms of asymptotic complexity.
- What you *have to show* are (relatively simple) *algebraic transformations* (mostly high school maths), e.g.
  - rules for exponentials,
  - rules for logarithms.
  - Therefore: Get your high school math knowledge reactivated if necessary!

## Order of Growth of Some Basic Functions / Growth Hierarchy

$\Theta(n^n)$

grows faster than (runs slower than)

$\Theta(n!)$

grows faster than (runs slower than)

$\Theta(k^n)$  for  $k > 1$

grows faster than

$\Theta(n^2)$

grows faster than

$\Theta(n)$

grows faster than

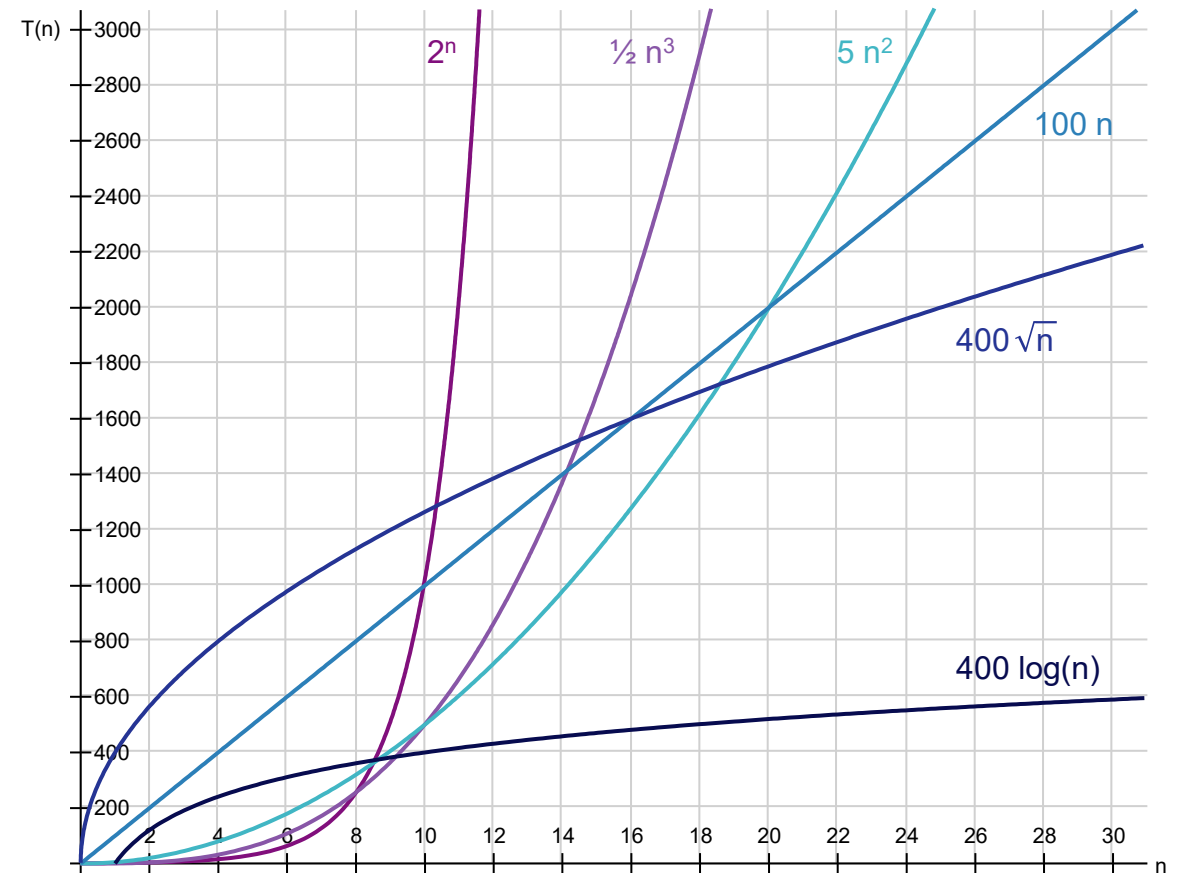
$\Theta(n^{1/2})$

grows faster than

$\Theta(\log(n))$

grows faster than

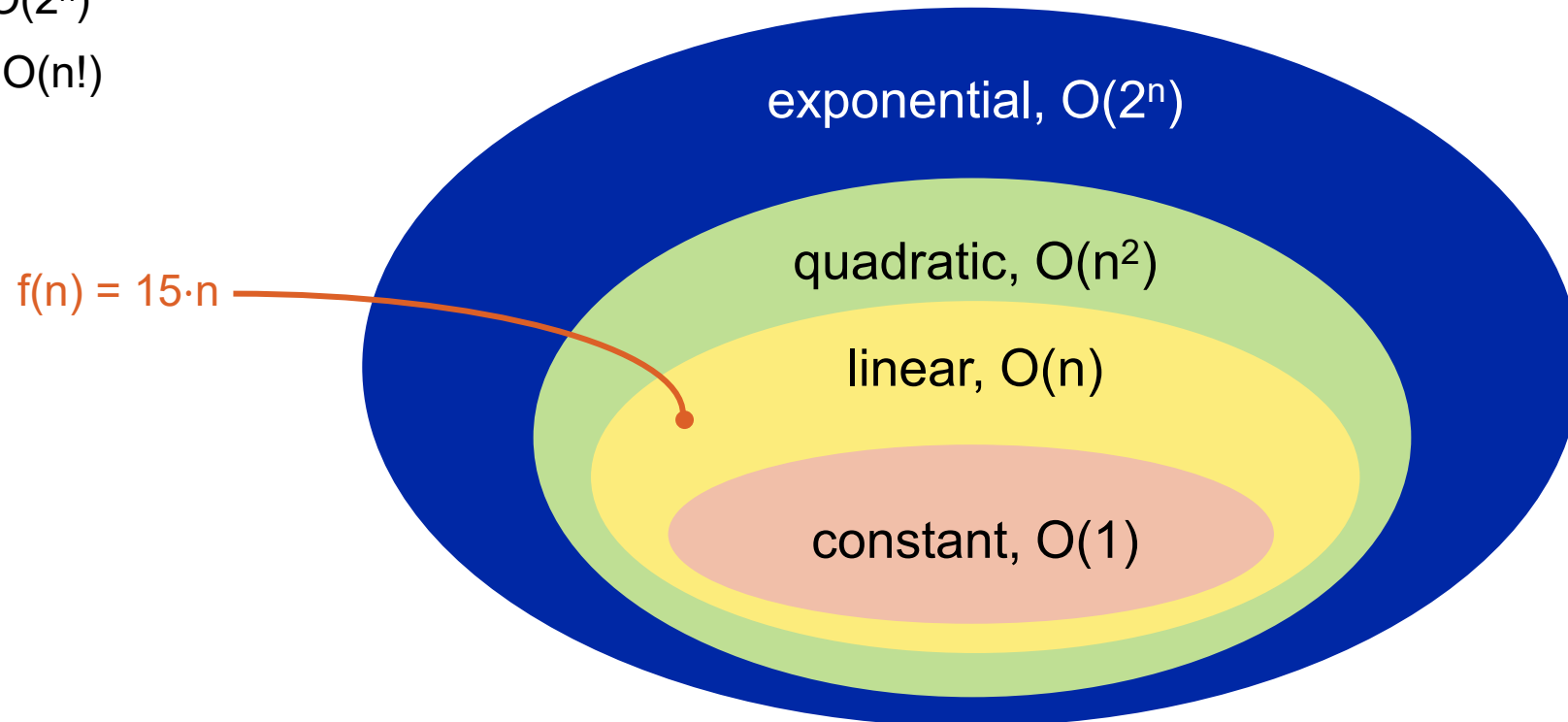
$\Theta(1)$



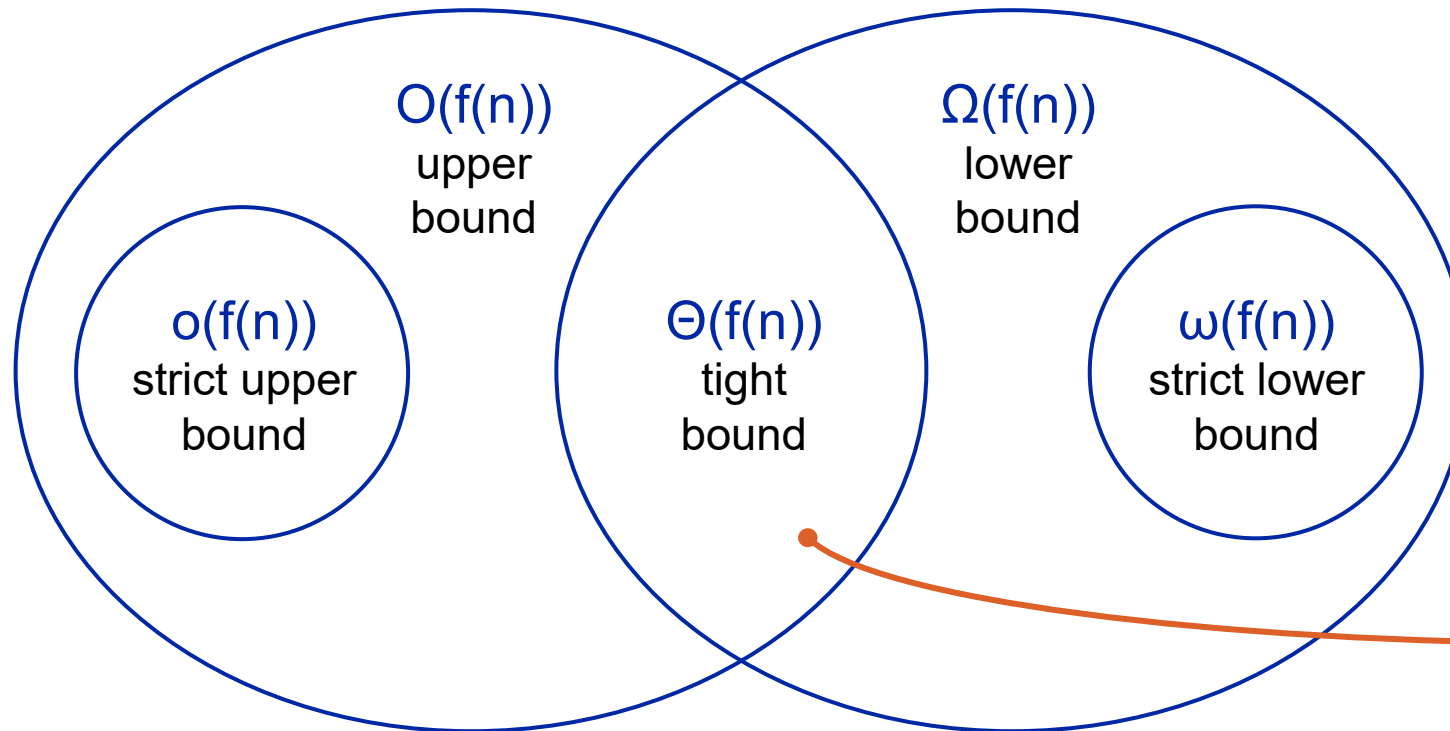
## Order of Growth of Some Basic Functions / Growth Hierarchy

From a perspective of Big O (non-tight bound), the sets of functions are contained within each other, e.g.:

- $f(n) = 15 \cdot n \in O(2^n)$
- $O(n \cdot \log(n)) \subsetneq O(n!)$



## Complexity Measure Types from a Set Perspective



$$\Theta(f(n)) \subsetneq O(f(n))$$

$$\Theta(f(n)) \subsetneq \Omega(f(n))$$

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$$

$$\omega(f(n)) \subsetneq \Omega(f(n))$$

$$o(f(n)) \subsetneq O(f(n))$$

e.g. with  $g(n) = 15 \cdot n$ ,  
 $g(n) \in O(n)$   
 $g(n) \in \Theta(n)$   
 $g(n) \in \Omega(n)$

## Additional Exercise: Complexity Classes as Sets of Functions

Which of the following statements are true and which are false?

- a)  $f(n) = 42n^2 + 40n + 2 \in O(n^2)$  ✓ true
- b)  $f(n) = 7^n \in O(5^n)$  ✗ false
- c)  $f(n) = 5^n \in \Theta(7^n)$  ✗ false
- d)  $f(n) = 5^n \in O(7^n)$  ✓ true
- e)  $f(n) = n^{10} \in \Omega(10^n)$  ✗ false
- f)  $f(n) = n^n \in \Omega(1)$  ✓ true
- g)  $O(\log(n)) \subsetneq O(\sqrt{n})$  ✓ true
- h)  $O(n) \subsetneq \Omega(n^2)$  ✗ false



## Algebraic Transformations: Exponentiation Rules

- (1)  $a^{-k} = 1 / a^k$
- (2)  $(a \cdot b)^k = a^k \cdot b^k$  and  $(a \div b)^k = a^k \div b^k$
- (3)  $a^m \cdot a^n = a^{(m+n)}$  and  $a^m \div a^n = a^{(m-n)} = a^m \cdot a^{-n}$
- (4)  $(a^m)^n = a^{(m \cdot n)}$
- (5)  $a^{1/k} = \sqrt[k]{a}$ , e.g.  $a^{3/4} = \sqrt[4]{a^3}$
- (6)  $a^0 := 1$  *regardless of the value of  $a$*

- Beware: in general:  $a^{(m^n)} \neq (a^m)^n = a^{(m \cdot n)}$   
 $a^{m^n}$  is by default to be read as  $a^{(m^n)}$
- Beware: in general:  $a^m + b^m \neq (a + b)^m$
- Beware: in general:  $(-a)^m \neq -a^m = -(a^m)$

## Algebraic Transformations: Logarithm Rules

(1)  $\log_b(m \cdot n) = \log_b(m) + \log_b(n)$

(2)  $\log_b(m \div n) = \log_b(m) - \log_b(n)$

(3)  $\log_b(m^n) = n \cdot \log_b(m)$  *regardless of the value of  $b$*

(4)  $a = b^{\log_b(a)}$

(5)  $\log_a(x) = \log_b(x) \div \log_b(a)$  *used for base transformation*

(6)  $a^{\log_b(x)} = x^{\log_b(a)}$  *regardless of the value of  $b$*

(7)  $\log_b(1) := 0$  *regardless of the value of  $b$*

Justification for (6):  $a^{\log_b(x)} \stackrel{(4)}{=} (b^{\log_b(a)})^{\log_b(x)} = b^{\log_b(a) \cdot \log_b(x)} = b^{\log_b(x) \cdot \log_b(a)} = x^{\log_b(a)}$

## Rules for and Some Notes on Asymptotic Complexity

Any addition of constants can be ignored:

$$\Theta(f(n) + c) = \Theta(f(n))$$

iff  $c = \text{const}$

*Example:*  $n^2$  is in  $O(n^2)$  and  $n^2 + 1000000000000$  is in  $O(n^2)$

*Remark:* In coding practice there are some constant additional factors which might not always be ignored, for example writing and reading from disk is usually very slow and might outweigh even complexity class unless for very large data sets.

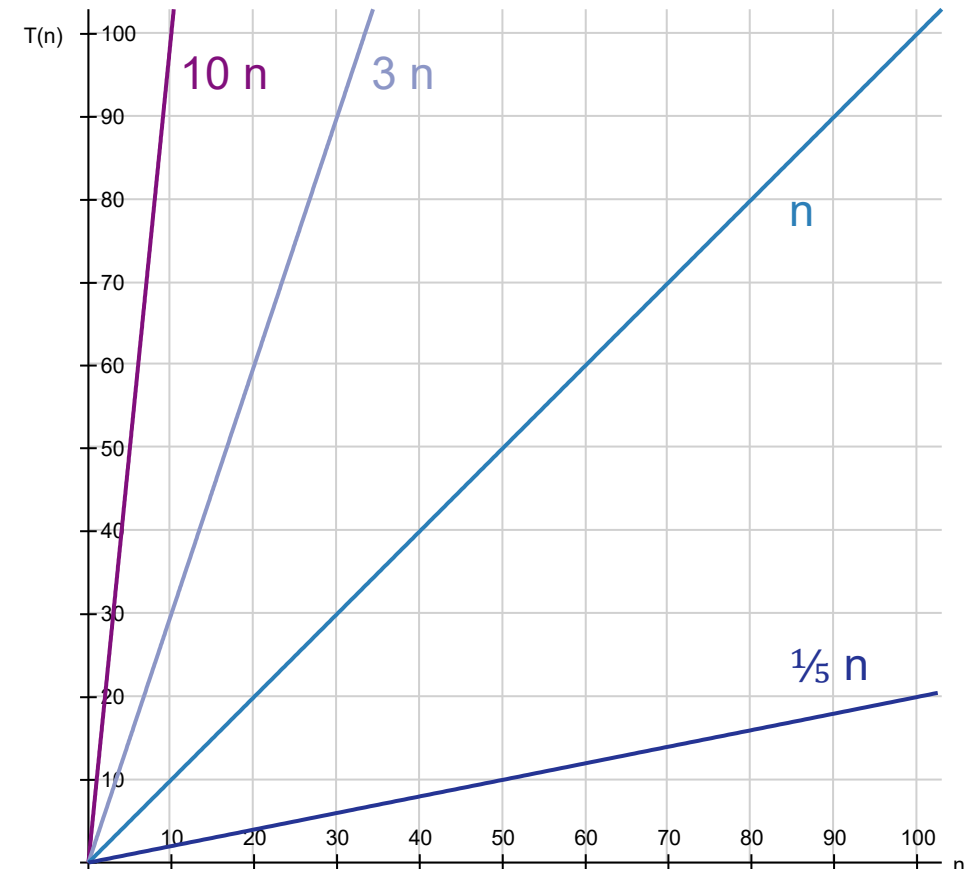
## Rules for and Some Notes on Asymptotic Complexity

Any multiplication with constants can be ignored:

$$\Theta(c \cdot f(n)) = \Theta(f(n))$$

iff  $c = \text{const}, c > 0$

*Example:*  $3n$  is in  $O(n)$  and  $99999999n$  is in  $O(n)$

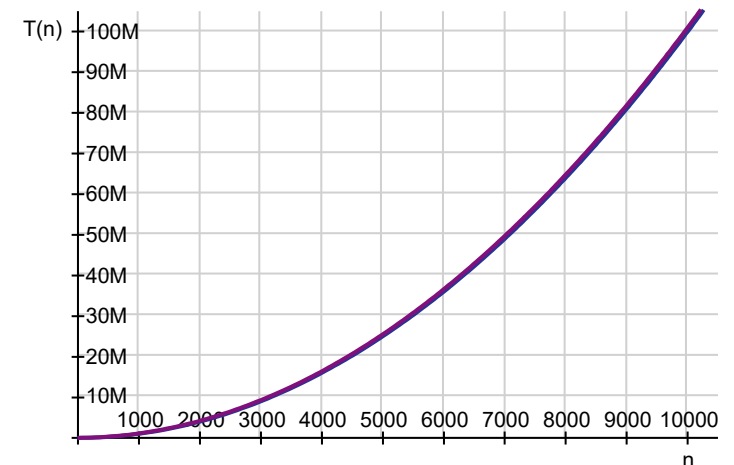
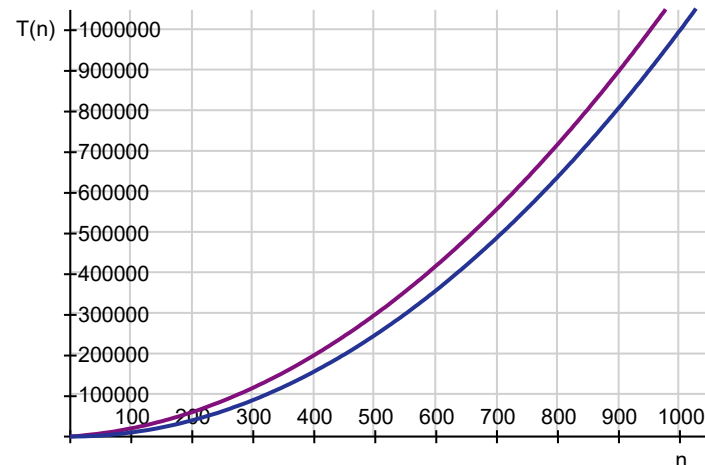
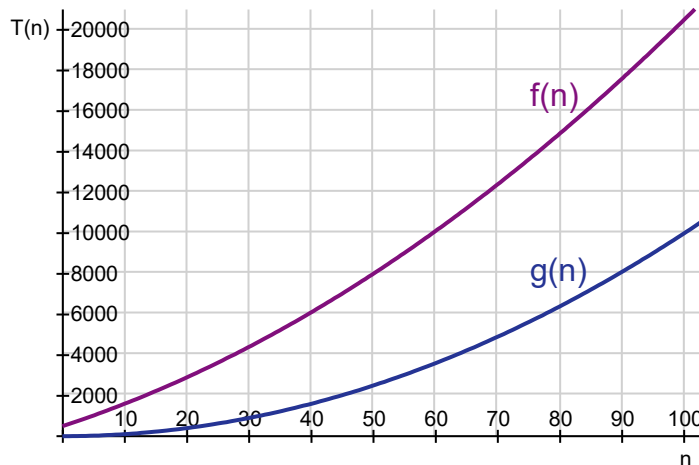


## Rules for and Some Notes on Asymptotic Complexity: Sums

Lower order terms get irrelevant asymptotically. In an sum of functions, the fastest growing function wins:

$$O(f(n) \pm g(n)) = O(\max(f(n), g(n)))$$

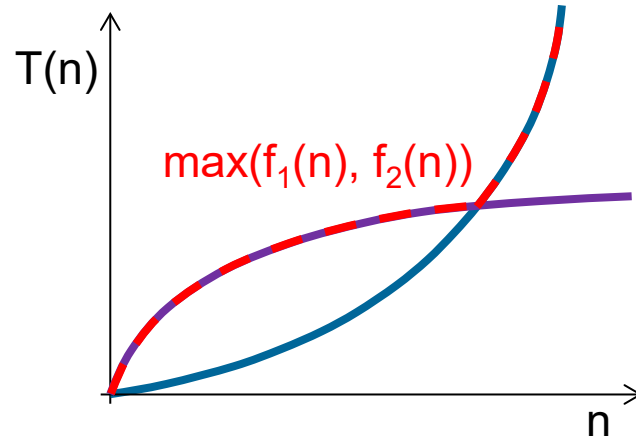
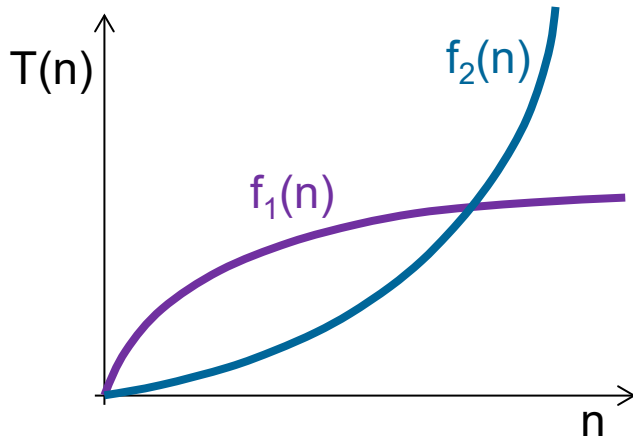
Comparison of  $f(n) = n^2 + 100n + 500$  and  $g(n) = n^2$



## Rules for and Some Notes on Asymptotic Complexity: Sums

The asymptotic complexity of the sum (or difference) of two functions is equal to the asymptotic complexity of the function summand which grows faster:

$$\Theta(f(n) \pm g(n)) = \Theta(\max(f(n), g(n))) \quad (\text{exception: } \Theta(f(n) - g(n)) \text{ in case } \Theta(f(n)) = \Theta(g(n)))$$



Another way to state this idea:  $\Theta(f(n) \pm g(n)) = \Theta(g(n))$  iff  $\Theta(g(n)) \geq \Theta(f(n))$

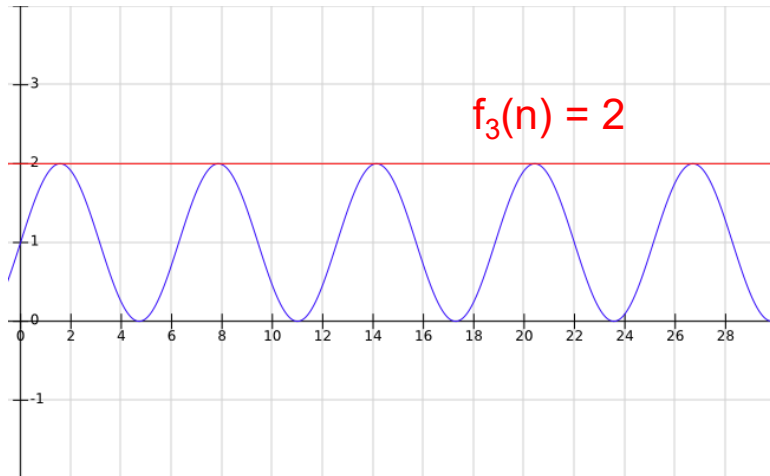
## Rules for and Some Notes on Asymptotic Complexity: Products

The asymptotic complexity of the product of two functions is:

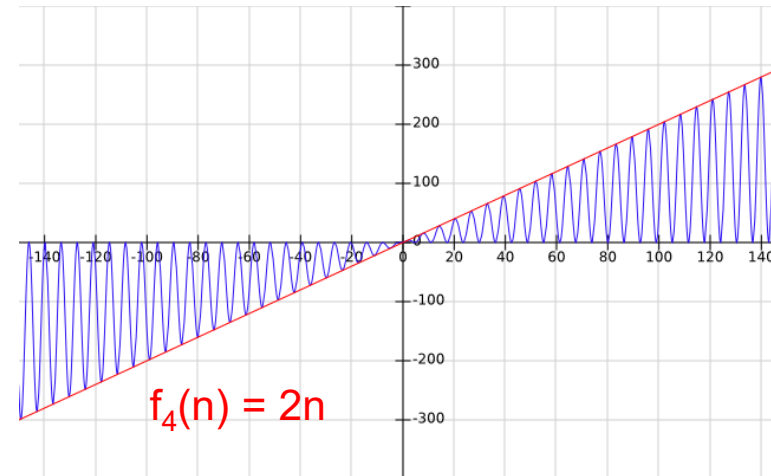
$$(f(n) \cdot g(n)) \in O(f(n) \cdot g(n))$$

*Example revisited:* What is the asymptotic complexity of the following functions?

$$f_1(n) = 1 + \sin(n) \rightarrow O(1)$$



$$f_2(n) = n \cdot (1 + \sin(n)) \rightarrow O(n)$$



## Rules for and Some Notes on Asymptotic Complexity: Polynomials

Polynomial family of functions, i.e. functions of type  $f(n) = n^k$ ,  $0 < k < \infty$ , are all distinct from each other and have growing order of growth with growing exponent («polynomial family of functions»).

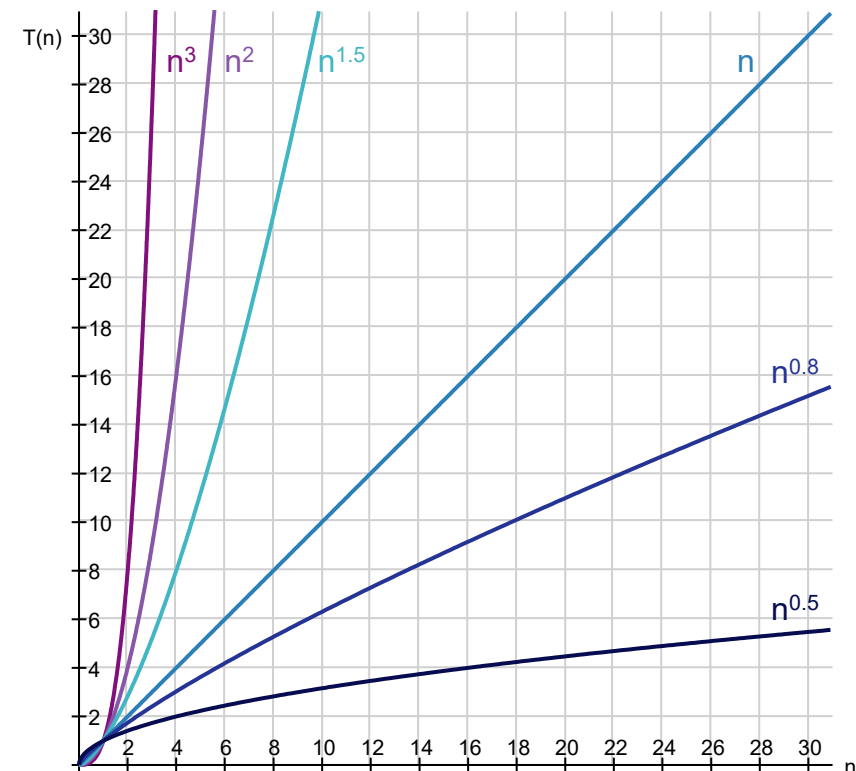
$$\Theta(n^x) \neq \Theta(n^y) \text{ iff } x \neq y$$

and

$$n^x \in O(n^y) \text{ iff } y \geq x$$

This also means, that the exponent of the root *does* matter:

$$\Theta(\sqrt[2]{n}) \neq \Theta(\sqrt[3]{n})$$



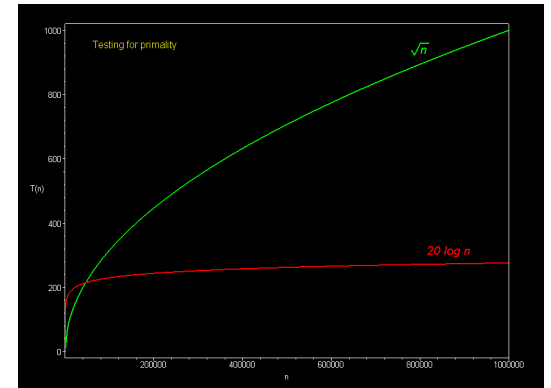
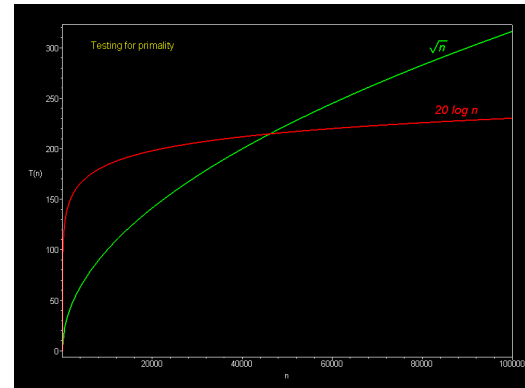
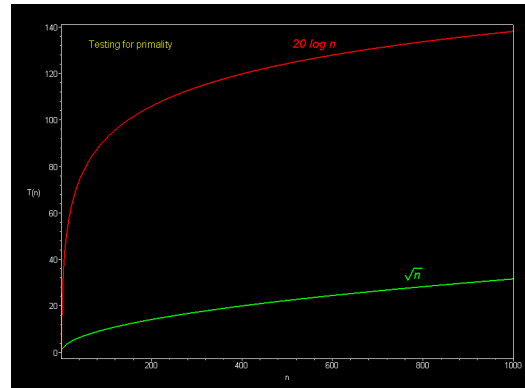
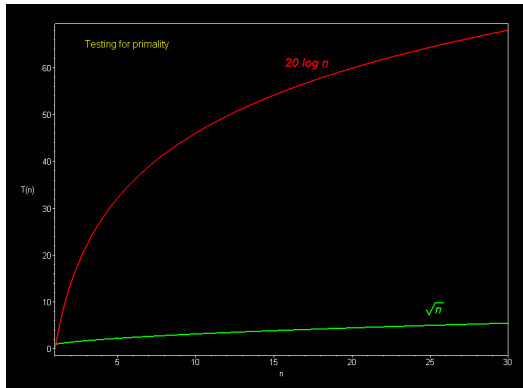


## Rules for and Some Notes on Asymptotic Complexity

Any polynomial function, i.e.  $n^k$ ,  $0 < k < \infty$ , regardless of exponent, grow faster than any logarithmic function.

*Example:*  $\sqrt{n}$  grows faster than  $\log(n)$  asymptotically

Comparison of  $f(n) = 20 \log(n)$  and  $g(n) = \sqrt{n}$



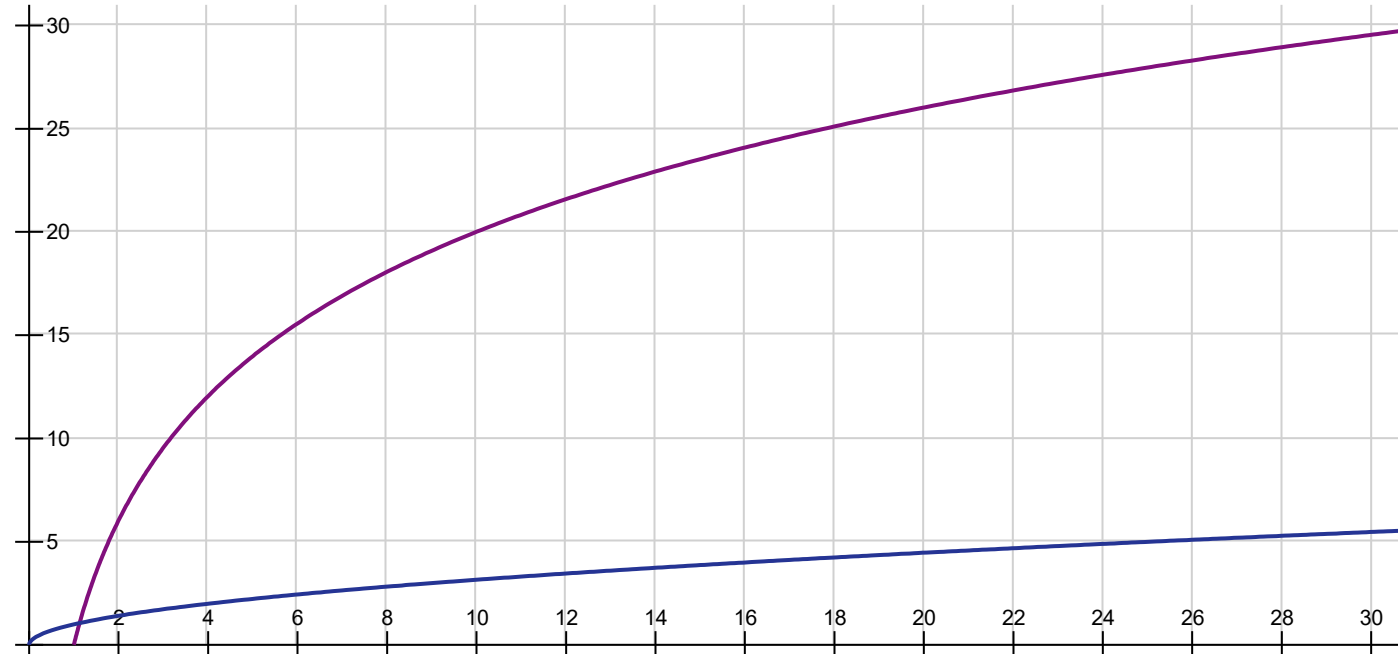
## Rules for and some notes on asymptotic complexity

Comparison of

$$f(n) = 20 \log(n)$$

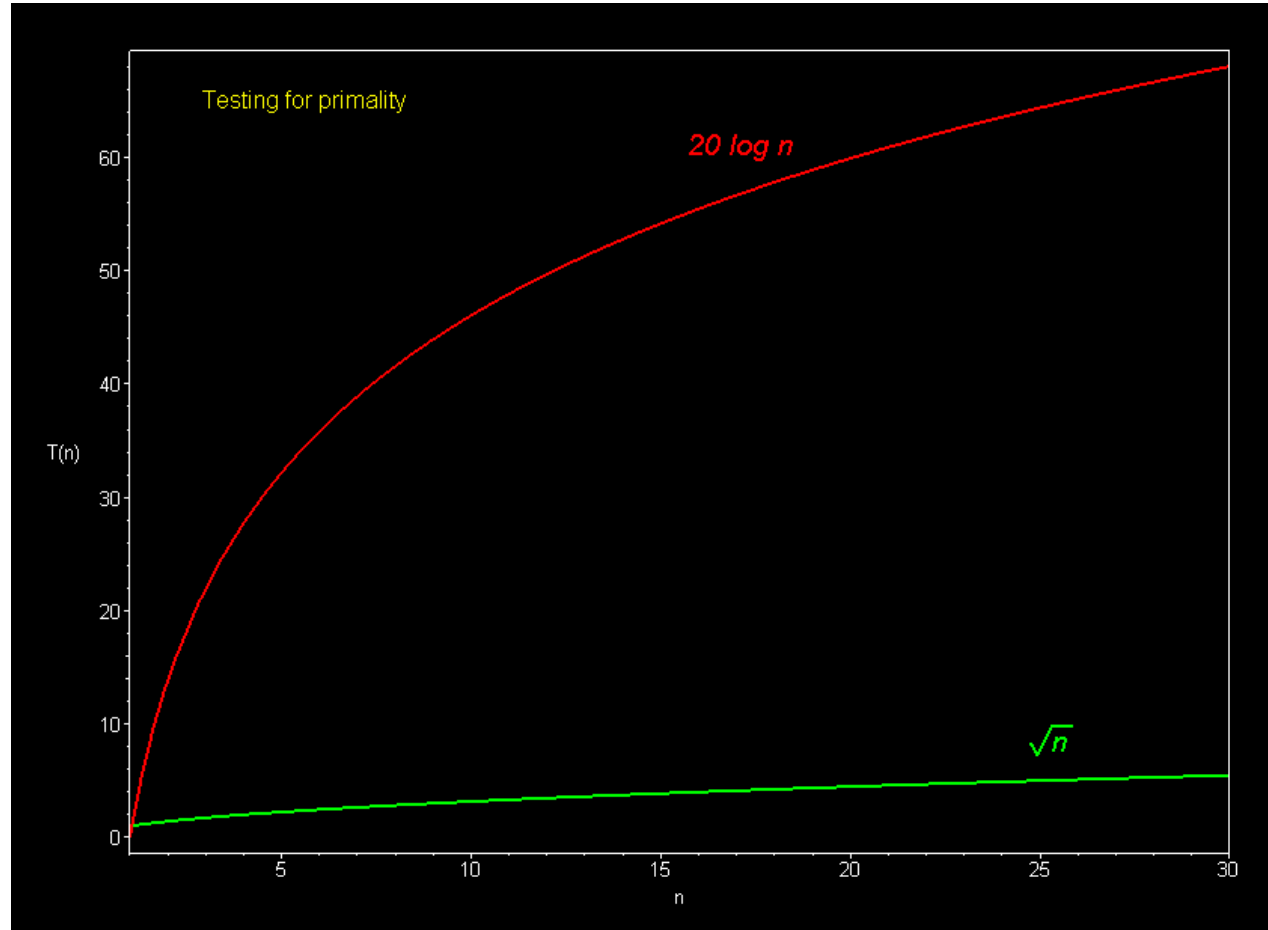
and

$$g(n) = \sqrt{n}$$



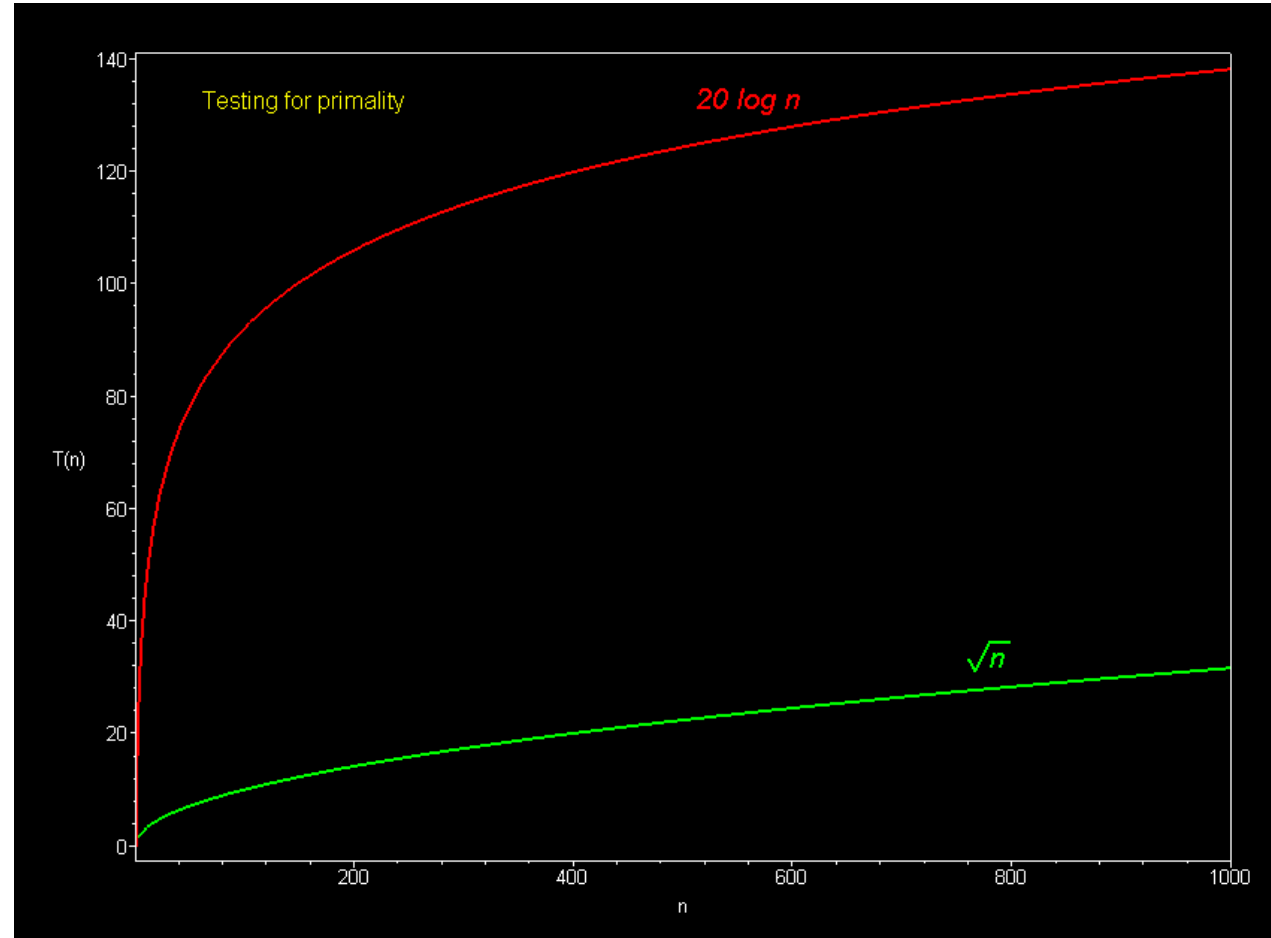
## Rules for and some notes on asymptotic complexity

Comparison of  
 $f(n) = 20 \log(n)$   
and  
 $g(n) = \sqrt{n}$



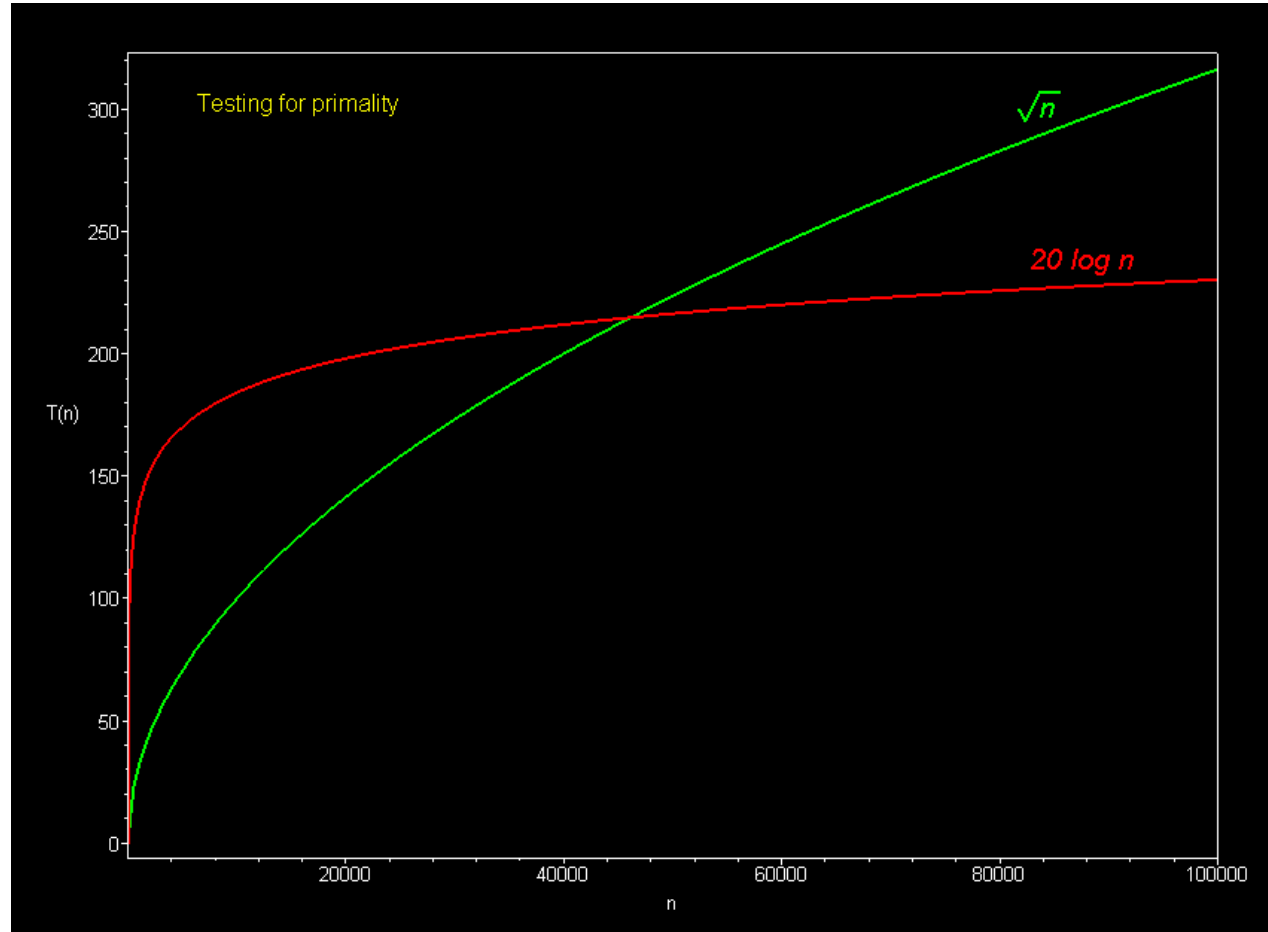
## Rules for and some notes on asymptotic complexity

Comparison of  
 $f(n) = 20 \log(n)$   
and  
 $g(n) = \sqrt{n}$



## Rules for and Some Notes on Asymptotic Complexity

Comparison of  
 $f(n) = 20 \log(n)$   
and  
 $g(n) = \sqrt{n}$



## Rules for and Some Notes on Asymptotic Complexity: Logarithms

- The base of logarithms doesn't matter for asymptotic complexity (as long as it is constant and isn't 1):

$$\begin{aligned}\Theta(\log_a(n)) &= \Theta(\log_b(n)) \\ &= \Theta(\lg(n)) = \Theta(\ln(n)) = \Theta(\text{ld}(n))\end{aligned}$$

Beware: The base of a logarithm is only irrelevant in this kind of situation. It is very relevant for example when  $f(n) = n^{\log_a(c)}$  is compared to  $g(n) = n^{\log_b(c)}$ .

- Potentially misleading notation:

$$\log^2(n) := \log(n) \cdot \log(n) = (\log(n))^2$$

and *not*, as one may probably expect,  $\log(\log(n))$ , i.e.  $\log^2(n) \neq \log(\log(n))$

- Also note that  $\log(n) \cdot \log(n)$  grows *faster* than  $\log(n)$  and  $\log(\log(n))$  grows *slower* than  $\log(n)$ .

## Rules for and Some Notes on Asymptotic Complexity: Exponentiation

- The basis of exponentiation *does* matter:

$$\Theta(a^n) \neq \Theta(b^n) \text{ iff } a \neq b$$

$$\text{e.g. } 3^n \notin \Theta(2^n) \text{ and } 3^n \notin O(2^n)$$

- Functions in  $O(n^n)$  grow faster than functions in  $O(n!)$  which grow faster than those in  $O(k^n)$  with  $k = \text{const}$  and  $k > 1$ .

*Justification:*

$$\begin{array}{lcl} k^n & = & k \cdot k \cdot k \cdot \dots \cdot k \\ n! & = & n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 1 \\ n^n & = & \underbrace{n \cdot n \cdot n \cdot \dots \cdot n}_{n \text{ times}} \end{array}$$

## Rules for and Some Notes on Asymptotic Complexity: Factorials

- Factorials grow exponentially (cf. Stirling formula).
- The logarithm of a factorial, i.e.  $\log(n!)$ , has the same asymptotic complexity as  $n \cdot \log(n)$ :

$$\log(n!) \in \Theta(n \cdot \log(n))$$

- Constant additions to a variable from which the factorials is taken *do matter*:

$$\Theta((n + k)!) \notin \Theta(n!)$$

Justification:  $(n + 1)! = n! \cdot (n + 1) = n \cdot n! + n! \in O(n \cdot n!) \neq O(n!)$

(in practice, though, it might be reasonable to just change the way how the input value  $n$  is defined...)



## Finding Asymptotic Complexity: Limes Method

It can be shown that for the limes superior of the fraction of two functions, the following equivalence is valid:

$$f(n) \in O(g(n)) \quad \Leftrightarrow \quad 0 \leq \limsup_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c < \infty$$

- Idea: Let the two functions «fight against each other» by building their fraction and look which one will asymptotically win.
- If the upper wins, the limes superior will go to infinity, if the lower wins, it will go to zero. If both functions belong to the same complexity class the limes superior will be some constant  $> 0$ .
- For finding the limes, l'Hospital's rule may also be helpful.
- Similar rules can be stated for the other kinds of asymptotic boundaries. For example, for  $\Theta$  notation use the limes instead of limes superior and distinguish between cases where it equals to zero, a constant or infinity.

## Limes Method Example

Consider the growth functions  $f(n) = \ln(n)$  and  $g(n) = \sqrt{n}$ . Which one grows faster?

We let  $f(n)$  and  $g(n)$  «fight against each other»:  
→ The fight has ended in a draw in first round...

$$\limsup_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \limsup_{n \rightarrow \infty} \frac{\ln(n)}{\sqrt{n}} = \frac{\infty}{\infty}$$

According to [l'Hospital's rule](#), we can write:

$$\begin{aligned} \limsup_{n \rightarrow \infty} \frac{f(n)}{g(n)} &= \limsup_{n \rightarrow \infty} \frac{\ln(n)}{\sqrt{n}} = \limsup_{n \rightarrow \infty} \frac{f'(n)}{g'(n)} = \\ &= \limsup_{n \rightarrow \infty} \frac{\frac{1}{n}}{\frac{1}{2} \cdot n^{-1/2}} = \limsup_{n \rightarrow \infty} \frac{2}{\sqrt{n}} = 0 < \infty \end{aligned}$$

Therefore:  $f(n) \in O(g(n))$  i.e.  $\ln(n) \in O(\sqrt{n})$  (and  $g(n)$  is even a strict upper bound of  $f(n)$ )

## Finding Asymptotic Complexity: Limes Method

Symbol	Meaning	Analogy	Limit Definition
$\mathcal{O}$	Upper bound	$f1 \leq f2$	$0 \leq \limsup_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c < \infty$
$\Omega$	Lower bound	$f1 \geq f2$	$0 < \liminf_{n \rightarrow \infty} \frac{f(n)}{g(n)} \leq \infty$
$\Theta$	Tight bound	$f1 = f2$	$0 < \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c < \infty$
$\mathcal{o}$	Strict upper bound	$f1 < f2$	$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$
$\omega$	Strict lower bound	$f1 > f2$	$\limsup_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$

## Rules for and Some Notes on Asymptotic Complexity

Note that **not every pair of functions can be compared** using Landau notation. Examples for not comparable functions include are:

- $f(n) = n^{0.5}$  and  $g(n) = n^{\sin(n)}$
- $f(n) = n$  and  $g(n) = \begin{cases} 1 & \text{if } n \text{ is odd} \\ n^2 & \text{if } n \text{ is even} \end{cases}$

## Asymptotic Complexity: Additional Examples

Find the asymptotic *tight* bounds ( $\Theta$  notation) of the following functions and rank them by their order of growth (slowest first).

- $f_1(n) = n^{10} \cdot \sqrt[4]{n}$
- $f_2(n) = \log(n^\pi \cdot \sqrt[2]{n})$
- $f_3(n) = n^{15} \cdot \sqrt[6]{n} \cdot \log(n)$
- $f_4(n) = \log_{\ln(5)}((\log(n))^{\lg(100)})$
- $f_5(n) = (\sqrt{2})^{\log_2(n)} + \log(n)$
- $f_6(n) = \log(n!) + n \cdot \log^5(n)$
- $f_7(n) = (\frac{2}{3})^n$

## Asymptotic Complexity: Additional Examples

Find the asymptotic *tight* bounds ( $\Theta$  notation) of the following functions and rank them by their order of growth (slowest first).

- $f_1(n) = n^{10} \cdot \sqrt[4]{n} \in \Theta(n^{10.25})$
- $f_2(n) = \log(n^\pi \cdot \sqrt[2]{n}) \in \Theta(\log(n))$
- $f_3(n) = n^{15} \cdot \sqrt[6]{n} \cdot \log(n) \in \Theta(n^{15\frac{1}{6}} \cdot \log(n))$
- $f_4(n) = \log_{\ln(5)}((\log(n))^{\lg(100)}) \in \Theta(\log(\log(n)))$
- $f_5(n) = (\sqrt{2})^{\log_2(n)} + \log(n) \in \Theta(n^{0.5})$
- $f_6(n) = \log(n!) + n \cdot \log^5(n) \in \Theta(n \cdot \log^5(n))$
- $f_7(n) = (\frac{2}{3})^n \in \Theta((\frac{2}{3})^n)$

*Ranking:* (slowest growing)  $f_7, f_4, f_2, f_5, f_6, f_1, f_3$  (fastest growing)

## Exercise 3 – Task 2: Asymptotic Complexity: Solution

$$f_1(n) = (2n + 3)! \quad \in \Theta((2n + 3)!)$$

$$f_2(n) = 2 \cdot \log(6^{\log(n^2)}) + \log(\pi \cdot n^2) + n^3 \quad \in \Theta(n^3)$$

$$f_3(n) = 4^{\log_2(n)} \quad \in \Theta(n^2)$$

$$f_4(n) = 12 \sqrt[2]{n} + 10^{223} + \log(5^n) \quad \in \Theta(n)$$

$$f_5(n) = 10^{\lg(20)} n^4 + 8^{229} n^3 + 20^{231} n^2 + 128n \cdot \log(n) \quad \in \Theta(n^4)$$

$$f_6(n) = \log(n^{2n+1}) \quad \in \Theta(n \cdot \log(n))$$

$$f_7(n) = 101^{\sqrt{n}} \quad \in \Theta(101^{\sqrt{n}})$$

$$f_8(n) = \log^2(n) + 50 \sqrt[2]{n} + \log(n) \quad \in \Theta(n^{0.5})$$

$$f_9(n) = n^n + 2^{2n} + 13^{124} \quad \in \Theta(n^n)$$

$$f_{10}(n) = 14400 \quad \in \Theta(1)$$

*Ranking:* (slowest growing)  $f_{10}, f_8, f_4, f_6, f_3, f_2, f_5, f_7, f_9, f_1$  (fastest growing)

## Exercise 3 – Task 2: Asymptotic Complexity: FAQ / Remarks

- Regarding  $f_1$ : Why is  $\Theta(n!) \neq \Theta((n + 3)!)$ ? Can't we just leave away the «+ 3»?
- Regarding  $f_4$ : Does «log» mean  $\log_2$ ,  $\log_{10}$  or  $\ln$  or something different?
- Regarding  $f_5$ : Does «lg» mean  $\log_2$ ,  $\log_{10}$  or  $\ln$  or something different?
- Regarding  $f_8$ : What does  $\log^2(n)$  mean?
- How can we show that  $f_1$  grows faster than  $f_9$ ?
- Regarding  $f_7$ : How do  $101^n$  and  $101^{\sqrt[7]{n^5}}$  compare to  $f_7$ ?
- If the task description would ask for  $O$  notation instead of  $\Theta$  notation: How would you solve the task in this case?



## Asymptotic Complexity: Additional Examples

- $f_1(n) = n \cdot \log_7(5^n)$
- $f_2(n) = 0.25 \cdot n^2 + (10 + \sin(n + 1.5)) \cdot n^{1.5} + 1169$
- $f_3(n) = n^\pi + n^3 + \sqrt[3]{n^{11}} + n^{2.5}$
- $f_4(n) = 7^{(n-5)}$
- $f_5(n) = \log_2(n) + \log_3(n) + \log_4(n) + \log_\pi(n)$
- $f_6(n) = 2^{2^n} + 2^{2n} + 4^n + 2^{n+1}$
- $f_7(n) = n^{(1 + 10^{-9})} + n \cdot \log(n) + \log(n!)$
- $f_8(n) = \max(\log^2(n), \log(\log(n))) + \log(\min(n, \sqrt{n}))$
- $f_9(n) = \log(\sum_{i=1}^n i)$
- $f_{10}(n) = 2^{16 \log_2(n^2)}$
- $f_{11}(n) = \sqrt{n} + n^{\frac{1}{\log(n)}} + \log(\sqrt{n})$



## Exercise 3, Past Exam Task

[2021 Final Exam] Assume  $f_1(n) = O(1)$ ,  $f_2(n) = O(N^2)$ , and  $f_3(n) = O(N \log N)$ . From these complexities it follows that  $f_1(n) + f_2(n) + f_3(n) = O(N \log N)$ .

Answer:

☐ True

☒ False



## Asymptotic Complexity: Additional Practice

Additional examples for practice can be found here:

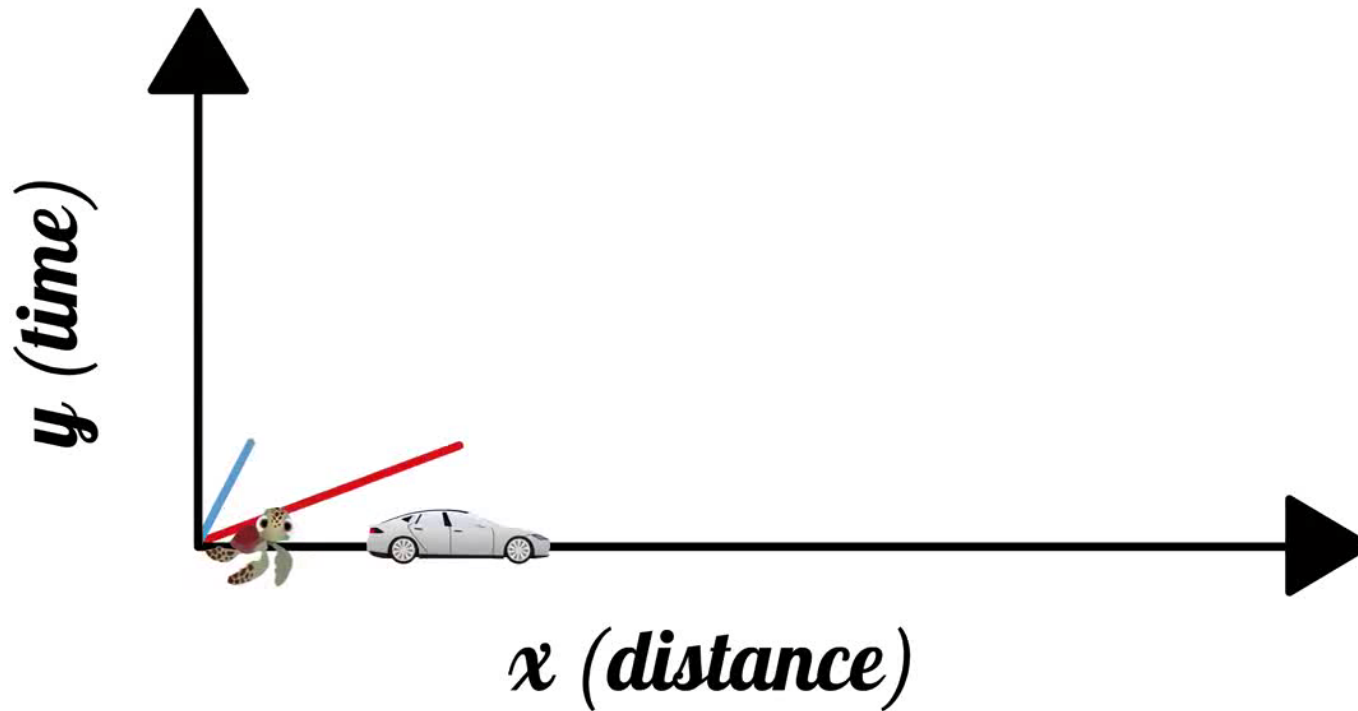
<https://h5p.org/node/455075>

## Another Perspective: Asymptotic Complexity as a Tool for Evaluating the Scaling Properties of a Problem

- What is the asymptotic complexity of the number of clinking sounds  $y = f(N)$  which are produced, when  $N$  persons are in a room and each person is clinking glasses with each other person exactly once?
- Given a network of server where each server is connected to every other server by wire (fully meshed topology): What is the asymptotic complexity of the number of wires required  $w = f(n)$  depending on the number of servers  $n$  in the network?
- What is the asymptotic complexity of the function  $v_{\max} = f(h)$  which gives the maximal velocity of a skydiver depending on the height  $h$  of the jump?
- What is the asymptotic complexity of a program which calculates the function  $f(h)$  defined above?

→ Asymptotic complexity is a tool to classify how well problems **scale**. We will use asymptotic complexity mostly to discuss how the execution time of algorithms scales with the input size. But more generally, the asymptotic complexity (or scaling properties) of problems are of importance in many situations outside computer science. The asymptotic complexity of an algorithm solving a particular problem in general is independent of the asymptotic complexity of the underlying problem.

## Asymptotic Complexity Illustrated



[https://www.youtube.com/watch?v=MyeV2\\_tGqvw](https://www.youtube.com/watch?v=MyeV2_tGqvw)

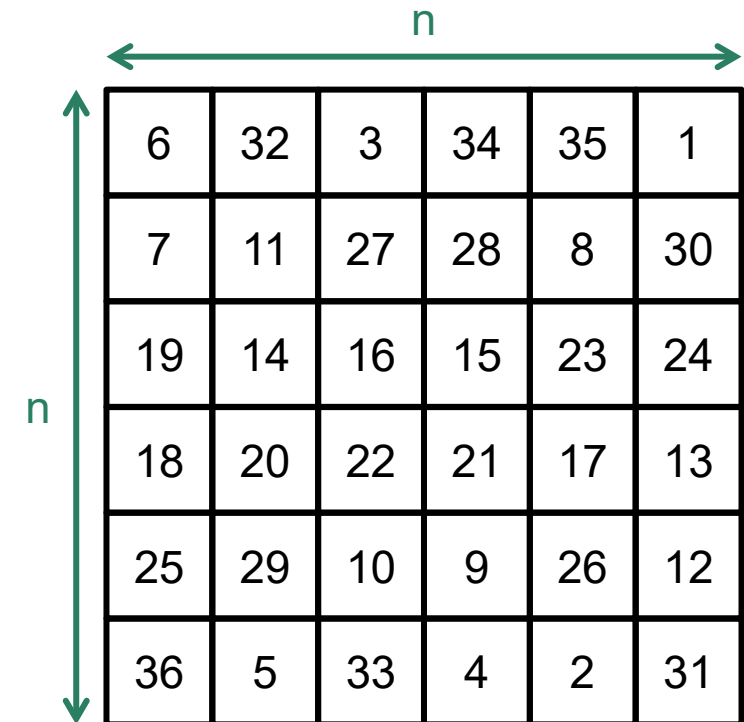
## Asymptotic Complexity: Defining the Input

Consider a square  $A[1..n, 1..n]$  of  $n$  times  $n$  integers with  $N = n \cdot n$  entries.

What is the asymptotic complexity of an algorithm which calculates the sum of the integers in  $A$ ?

→ Depending on what you consider the variable with regard to which the asymptotic complexity is specified, it could either be written as  $O(n^2)$  or as  $O(N)$ .

It is therefore quite important to always clearly specify what the input is to which you refer.



6	32	3	34	35	1
7	11	27	28	8	30
19	14	16	15	23	24
18	20	22	21	17	13
25	29	10	9	26	12
36	5	33	4	2	31



## Running Time Analysis of Code Snippets

- Try to get a «gut feeling» about asymptotic complexity of a code by just looking at it without a lengthy in-depth analysis.
- This may also be helpful to better understand asymptotic complexity.



## Running Time Analysis: Example a)

In which asymptotic complexity class is the following C code function?

```
void functionA(int n)
{
    for (int i = n; i > 1; i -= 2)
    {
        printf(i);
    }
}
```

→  **$O(n)$**



## Running Time Analysis: Example b)

In which asymptotic complexity class is the following C code function?

```
void functionB(int n)
{
    for (int i = 1; i < n; i++)
    {
        for (int j = n; j > n - i; j--)
        {
            printf(j);
        }
    }
}
```

→  $O(n^2)$

## Running Time Analysis: Example c)

In which asymptotic complexity class is the following C code function?

```
void functionC(int n)
{
    for (int i = n; i > 1; i = i / 2)
    {
        for (int j = 1; j <= n; j++)
        {
            printf("Hello World!");
        }
    }
}
```

The outer for loop has an asymptotic complexity of  $O(\log(n))$ , the inner of  $O(n)$ , in total there is a complexity of  **$O(n \cdot \log(n))$** .

## Running Time Analysis: Example d)

In which asymptotic complexity class is the following C code function?

```
int functionD(int n) {  
    int z = 0;  
    for (int i = 0; i < n; i++) {  
        for (int j = 0; j < n; j++) {  
            for (int k = 0; k < j; k++) {  
                z = z + 1;  
            }  
        }  
    }  
    return z;  
}
```

→  $O(n^3)$

## Running Time Analysis: Example e)

In which asymptotic complexity class is the following C code function?

```
void functionE(int n)
{
    for (int i = n; i <= n; i++)
    {
        for (int j = n; j > 1; j = j / 2)
        {
            printf(j);
        }
    }
}
```

The outer for loop will be executed only exactly once (since the counter variable is initiated to  $n$  which is the upper boundary for the loop), the inner loop has an asymptotic complexity of  $O(\log(n))$ , in total there is a complexity of  **$O(\log(n))$** .

## Running Time Analysis: Example f)

In which asymptotic complexity class is the following C code function?

```
void functionF(int n)
{
    for (int i = 1; i <= n * n; i += 10)
    {
        for (int j = 1; j * j <= n; j++)
        {
            printf("Hello World!");
        }
    }
}
```

The inner loop will be passed through  $\lceil \sqrt{n} \rceil$  times, the outer loop  $n^2/10$  times.

This can also be seen by drawing the square root of all the variables used in the loop conditions.

$$\rightarrow O(n^2 \cdot \sqrt{n}) = O(n^{2.5})$$

## Running Time Analysis: Example g)

What is the asymptotic bound for the following C code fragment?

```
int functionG(int n)
{
    int z = 42;
    for (int i = 1; i < n; i++)
    {
        for (int j = 1; j < n * n + 1; j++)
        {
            return z;
        }
    }
}
```

The function will always abort and return when it reaches the innermost loop body for the first time. Therefore the time needed will not depend on the size of  $n$ .

→  **$O(1)$**



## Running Time Analysis: Additional Exercise

- a) Write a function in C code which has a asymptotic running time in  $\Theta(n^{3.5} \cdot \log(n))$ .
- b) Write a function in C code which has a asymptotic running time in  $\Theta(2^n)$ .
- c) Write a function in C code which has a asymptotic running time in  $\Theta(n!)$ .

## Asymptotic Complexity: Potential Misconception

Where is the mistake in the following statement?

«If an algorithm is given which is known to be of (tight) linear complexity, i.e. is in  $\Theta(c \cdot n)$  where  $c$  is a constant factor, and we know that this algorithm takes 6 seconds to process an input of size  $n = 100$ , then this algorithm will take about 60 seconds to process an input of size  $n = 1000$ .»

Algorithm: example(n)	Cost	Number of times executed	
$x = n + 42$	10 ns	1	
$x = 3 \cdot x^5 + 7 \cdot x^3$	50 ns	1	
<b>for</b> $i = 1$ <b>to</b> $2 \cdot n$ <b>do</b>	20 ns	$n + 1$	
$x = x + i$	10 ns	$n$	
print(x)	1000 ns	1	

Time needed for  $n = 10$ :

$$T(10) = 1060 \text{ ns} + 320 \text{ ns} = 1380 \text{ ns}$$

Time needed for  $n = 100$ :

$$T(100) = 1060 \text{ ns} + 3200 \text{ ns} = 4260 \text{ ns}$$

$$T(100) \neq 2 \cdot 10 \cdot T(10)$$





# Algorithm Running Time Analysis

- General Principles
- Examples Solved

## Running Time Analysis of Loops: Example I

**Algorithm:** my\_algo(n)

```
1  for i = 1 to n do  
2      x = 0  
3      a = 54  
4      for a to 721 do  
5          x = x + 1
```

## Running Time Analysis: for Loops

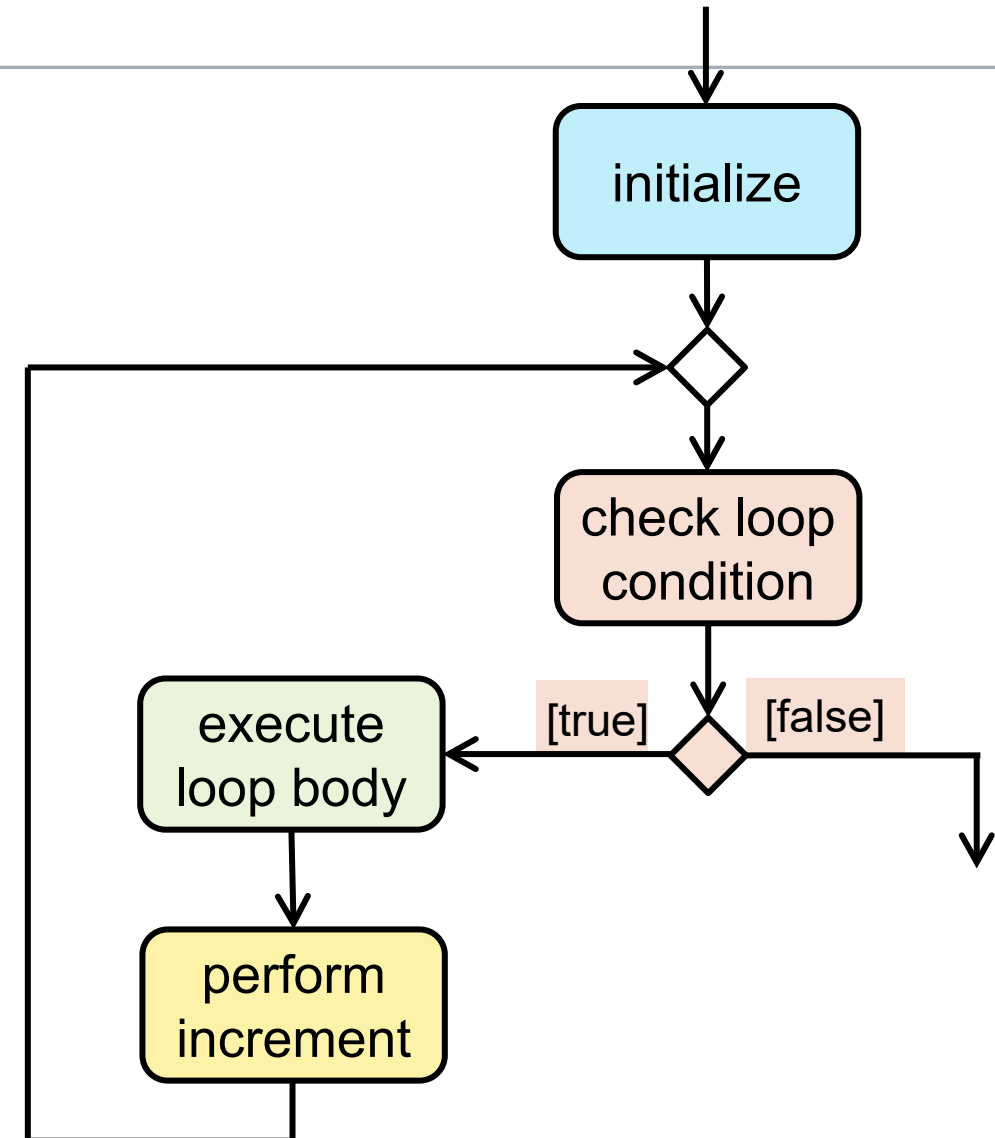
A for loop consists of an **header** and a **body**. The header has three parts (separated by semicolons in C code): **initialization**, **condition** and **increment**.

The initialization is executed exactly once.

The loop exit condition check is executed...

- at least once,
- one more time than the body of the loop and the increment.

```
for ( /*ini*/; /*con*/; /*inc*/ ) {
    /* loop body */
}
```



## Running Time Analysis: Loops: Growth Rate and Step Size

The **increment** of a loop can have different **growth rates**, for example constant (e.g.  $i = i + 1$ ), linear (e.g.  $i = 5 * i$ ), quadratic (e.g.  $i = i * i$ ) etc. **Usually**, the increment is a **constant** which is called the **step size**.

Loops with a constant increment (step size) can...

- be **counting up** (e.g.  $i = i + 1$ ,  $i = i + 2$ , ...) or **counting down** (e.g.  $i = i - 1$ ,  $i = i - 2$ , ...), and
- have a **step size of 1** («counter») or a step size **bigger than 1**.

*Example of a down-counting for loop with a step size of 42:*

```
for (i = 1000; i > 0; i = i - 42) {  
    /* loop body */  
}
```

## Running Time Analysis: Up-Counting Loop With Step Size 1

The number of executions of the **body** of a canonical\* **up-counting for loop with a step size of 1**, regardless what the start and end values are, can be calculated as follows, assuming that the **upper boundary** of the condition is **inclusive** (i.e. comparison using  $\leq$  operator):

**«end – start + 1»** and once more for the header

Examples:

<i>START</i>	<i>END</i>	<i>Number of times executed</i>
<b>for</b> $i = 1$ <b>to</b> $n$ <b>do</b>		$n + 1 = \text{END} - \text{START} + 2 = n - 1 + 2 = n + 1$
<loop body>		$n$

<i>START</i>	<i>END</i>	<i>Number of times executed</i>
<b>for</b> ( $i = 258$ ; $i \leq 937$ ; $i++$ ) {		$914 = \text{END} - \text{START} + 2 = 937 - 258 + 2 = 681$
/* loop body */		680
}		

## Running Time Analysis: Down-Counting Loop With Step Size 1

In case of a **down-counting** loop with **step size 1**, the number of executions of the **body** is:

**«start – end + 1»** and once more for the header

Examples:

<i>START</i> <i>END</i>	<i>Number of times executed</i>
<b>for</b> <i>i</i> = <i>n</i> <b>to</b> <i>1</i> <b>do</b>	$n + 1 = \text{START} - \text{END} + 2 = n - 1 + 2 = n + 1$
<loop body>	<i>n</i>

<i>START</i> <i>END</i>	<i>Number of times executed</i>
<b>for</b> ( <i>i</i> = 87; <i>i</i> >= 17; <i>i</i> ++) {	$72 = \text{START} - \text{END} + 2 = 87 - 17 + 2 = 72$
/* loop body */	71
}	

## Running Time Analysis: Loop With Arbitrary Fixed Step Size

In case of a (well-behaved) for loop with an **arbitrary** (but constant and integer) **step size**, the number of executions of the body can be calculated as follows:

*up-counting:* 
$$\left\lfloor \frac{\text{end} - \text{start} + \text{step size}}{\text{step size}} \right\rfloor$$

*down-counting:* 
$$\left\lfloor \frac{\text{start} - \text{end} + \text{step size}}{\text{step size}} \right\rfloor$$

Examples:

<i>START</i>	<i>END</i>	<i>STEP</i>	<i>Number of times executed</i>
for i = 5	to n	step 5	$\lfloor n / 5 \rfloor + 1$
do			$\left\lfloor \frac{\text{END} - \text{START} + \text{STEP}}{\text{STEP}} \right\rfloor = \lfloor n / 5 \rfloor$
[ <loop body>			

<i>START</i>	<i>END</i>	<i>STEP</i>	<i>Number of times executed</i>
for (i = 57;	i >= 23;	i = i - 3)	{ 13
/* loop body */			12
}			

## Running Time Analysis: Overview of Formulas for Calculation of Number of Executions

<i>counting direction</i>	<i>boundary inclusive?</i>	<i>step size</i>	<i>code example</i>	<i>no. of executions of the loop body</i>	<i>no. of executions of the loop header</i>
up	yes	1	<code>for(i = 1; i &lt;= 99; i++)</code>	end – start + 1	end – start + 2
down	yes	1	<code>for(i = 99; i &gt;= 1; i--)</code>	start – end + 1	start – end + 2
up	no	1	<code>for(i = 1; i &lt; 99; i++)</code>	end – start	end – start + 1
down	no	1	<code>for(i = 99; i &gt; 1; i--)</code>	start – end	start – end + 1
up	yes	arbitrary	<code>for(i = 1; i &lt;= 99; i=i+3)</code>	$\left\lfloor \frac{\text{end} - \text{start} + \text{step}}{\text{step}} \right\rfloor$	$\left\lfloor \frac{\text{end} - \text{start} + \text{step}}{\text{step}} \right\rfloor + 1$
down	yes	arbitrary	<code>for(i = 99; i &gt;= 1; i=i-7)</code>	$\left\lfloor \frac{\text{start} - \text{end} + \text{step}}{\text{step}} \right\rfloor$	$\left\lfloor \frac{\text{start} - \text{end} + \text{step}}{\text{step}} \right\rfloor + 1$



## Running Time Analysis of Loops: Example II

**Algorithm:** my\_algo( $n$ )

```
1 for  $i = 2$  to  $n - 1$  do  
2   print( $i$ )  
3   for  $j = i + 1$  to  $n - i$  do  
4     print( $j$ )
```



## Methods for Determining the Number of Executions of for Loops in Asymptotic Running Time Analysis

- Multiplicative combination of algorithm parts (and using «end – start + 1» et al.)
- «Table method»: write down the evolution of the loop variable in a table to gain an understanding about the number of executions (probably not possible for more than two levels of nesting)
- Write loop as a nested multiple sum and solve it (requires a bit of knowledge how to manipulate sums).

# Number of Executions of the Body of Loops

	<i>pseudo code fragment</i>	<i>table visualization</i>	<i>«end-start+1» summation</i>	<i>pure summation</i>																								
<i>Independent loops</i>	<pre>for i = 1 to n do   for j = 1 to n do     j = j + 1</pre>	<table><tr><th>i =</th><th>j =</th><th>executions</th></tr><tr><td>1</td><td>1 ... n</td><td>n</td></tr><tr><td>2</td><td>1 ... n</td><td>n</td></tr><tr><td>⋮</td><td>⋮ ⋮ ⋮</td><td>⋮</td></tr><tr><td>k</td><td>1 ... n</td><td>n</td></tr><tr><td>⋮</td><td>⋮ ⋮ ⋮</td><td>⋮</td></tr><tr><td>n-1</td><td>1 ... n</td><td>n</td></tr><tr><td>n</td><td>1 ... n</td><td>n</td></tr></table>	i =	j =	executions	1	1 ... n	n	2	1 ... n	n	⋮	⋮ ⋮ ⋮	⋮	k	1 ... n	n	⋮	⋮ ⋮ ⋮	⋮	n-1	1 ... n	n	n	1 ... n	n	$\sum_{i=1}^n \text{end} - \text{start} + 1 =$ $= \sum_{i=1}^n n - 1 + 1 = \sum_{i=1}^n n = n \cdot n = n^2$	$\sum_{i=1}^n \sum_{j=1}^n 1 =$ $= n^2$
i =	j =	executions																										
1	1 ... n	n																										
2	1 ... n	n																										
⋮	⋮ ⋮ ⋮	⋮																										
k	1 ... n	n																										
⋮	⋮ ⋮ ⋮	⋮																										
n-1	1 ... n	n																										
n	1 ... n	n																										
<i>Loops connected through initialization</i>	<pre>for i = 1 to n do   for j = i + 1 to n do     j = j + 1</pre>	<table><tr><th>i =</th><th>j =</th><th>executions</th></tr><tr><td>1</td><td>2 ... n</td><td>n - 1</td></tr><tr><td>2</td><td>3 ... n</td><td>n - 2</td></tr><tr><td>⋮</td><td>⋮ ⋮ ⋮</td><td>⋮</td></tr><tr><td>k</td><td>k + 1 ... n</td><td>n - k</td></tr><tr><td>⋮</td><td>⋮ ⋮ ⋮</td><td>⋮</td></tr><tr><td>n-1</td><td>n ... n</td><td>1</td></tr><tr><td>n</td><td>n + 1 ... n</td><td>0</td></tr></table>	i =	j =	executions	1	2 ... n	n - 1	2	3 ... n	n - 2	⋮	⋮ ⋮ ⋮	⋮	k	k + 1 ... n	n - k	⋮	⋮ ⋮ ⋮	⋮	n-1	n ... n	1	n	n + 1 ... n	0	$\sum_{i=1}^n \text{end} - \text{start} + 1 =$ $= \sum_{i=1}^n n - (i + 1) + 1 = \sum_{i=1}^n n - i = n^2 - \frac{n \cdot (n + 1)}{2}$	$\sum_{i=1}^n \sum_{j=i+1}^n 1 =$ $= n^2 - \frac{n \cdot (n + 1)}{2}$
i =	j =	executions																										
1	2 ... n	n - 1																										
2	3 ... n	n - 2																										
⋮	⋮ ⋮ ⋮	⋮																										
k	k + 1 ... n	n - k																										
⋮	⋮ ⋮ ⋮	⋮																										
n-1	n ... n	1																										
n	n + 1 ... n	0																										
<i>Loops connected through stop condition</i>	<pre>for i = 1 to n do   for j = 1 to n - i do     j = j + 1</pre>	<table><tr><th>i =</th><th>j =</th><th>executions</th></tr><tr><td>1</td><td>1 ... n-1</td><td>n - 1</td></tr><tr><td>2</td><td>1 ... n-2</td><td>n - 2</td></tr><tr><td>⋮</td><td>⋮ ⋮ ⋮</td><td>⋮</td></tr><tr><td>k</td><td>1 ... n-k</td><td>n - k</td></tr><tr><td>⋮</td><td>⋮ ⋮ ⋮</td><td>⋮</td></tr><tr><td>n-1</td><td>1 ... 1</td><td>1</td></tr><tr><td>n</td><td>1 ... 0</td><td>0</td></tr></table>	i =	j =	executions	1	1 ... n-1	n - 1	2	1 ... n-2	n - 2	⋮	⋮ ⋮ ⋮	⋮	k	1 ... n-k	n - k	⋮	⋮ ⋮ ⋮	⋮	n-1	1 ... 1	1	n	1 ... 0	0	$\sum_{i=1}^n \text{end} - \text{start} + 1 =$ $= \sum_{i=1}^n (n - i) - 1 + 1 = \sum_{i=1}^n n - i = n^2 - \frac{n \cdot (n + 1)}{2}$	$\sum_{i=1}^n \sum_{j=1}^{n-i} 1 =$ $= n^2 - \frac{n \cdot (n + 1)}{2}$
i =	j =	executions																										
1	1 ... n-1	n - 1																										
2	1 ... n-2	n - 2																										
⋮	⋮ ⋮ ⋮	⋮																										
k	1 ... n-k	n - k																										
⋮	⋮ ⋮ ⋮	⋮																										
n-1	1 ... 1	1																										
n	1 ... 0	0																										



## Methods for Determining the Number of Executions of for Loops in Asymptotic Running Time Analysis

- Multiplicative combination of algorithm parts (and using «end – start + 1» et al.)
- «Table method»: write down the evolution of the loop variable in a table to gain an understanding about the number of executions (probably not possible for more than two levels of nesting)
- Write loop as a nested multiple sum and solve it (requires a bit of knowledge how to manipulate sums).

## Algorithm Running Time Analysis: Example 1

*Task 1 of assignment 2 from spring semester 2016:*

Given an unsorted array  $A[1..n]$  of integers and an integer  $k$ , the adjacent algorithm calculates the maximum value of every contiguous subarray of size  $k$ .

Perform an exact analysis of the running time of the following algorithm.

Also determine the best and the worst case of the algorithm and what the running time and asymptotic complexity is in each case.

**Algorithm:**  $\text{findKmax}(A, k, n)$

---

```
for  $i = 1$  to  $n - k + 1$  do
     $\text{max} = A[i]$ 
    for  $j = 1$  to  $k - 1$  do
        if  $A[i + j] > \text{max}$  then
             $\text{max} = A[i + j]$ 
    print( $\text{max}$ )
```

## Algorithm Running Time Analysis: Example 1 Solution

Algorithm: findKmax(A, k, n)	Cost	Number of times executed
1 <b>for</b> i = 1 <b>to</b> n - k + 1 <b>do</b>	$c_1$	$n - k + 2$
2     max = A[i]	$c_2$	$n - k + 1$
3 <b>for</b> j = 1 <b>to</b> k - 1 <b>do</b>	$c_3$	$k \cdot (n - k + 1)$
4 <b>if</b> A[i + j] > max <b>then</b>	$c_4$	$(k - 1) \cdot (n - k + 1)$
5             max = A[i + j]	$c_5$	$\alpha \cdot (k - 1) \cdot (n - k + 1); \quad 0 \leq \alpha \leq 1$
6     print(max)	$c_6$	$n - k + 1$

$$\begin{aligned}
 T(n) &= c_1 \cdot (n - k + 2) + c_2 \cdot (n - k + 1) + c_3 \cdot k \cdot (n - k + 1) + c_4 \cdot (k - 1) \cdot (n - k + 1) + \\
 &\quad + c_5 \cdot \alpha \cdot (k - 1) \cdot (n - k + 1) + c_6 \cdot (n - k + 1) = \\
 &= c_1 + (c_1 + c_2 - c_4 + c_6 - c_5 \cdot \alpha + (c_3 + c_4 + c_5 \cdot \alpha) \cdot k) \cdot (n - k + 1) \quad \rightarrow \text{in worst case } O(n^2)
 \end{aligned}$$

## Algorithm Running Time Analysis: Example 2

*Exercise 2.3 from midterm 1 of spring semester 2016:*

Perform an exact analysis of the running time of the following algorithm and determine its asymptotic complexity.

**Algorithm:**  $\text{alg}(A, n)$

---

```
1  for  $i = 1$  to  $\lfloor n / 2 \rfloor$  do
2       $\text{min} = i$ 
3       $\text{max} = n - i + 1$ 
4      if  $A[\text{min}] > A[\text{max}]$  then
5           $\text{exchange } A[\text{min}] \text{ and } A[\text{max}]$ 
6      for  $j = i + 1$  to  $n - i$  do
7          if  $A[j] < A[\text{min}]$  then
8               $\text{min} = j$ 
9          if  $A[j] > A[\text{max}]$  then
10              $\text{max} = j$ 
11      $\text{exchange } A[i] \text{ and } A[\text{min}]$ 
12      $\text{exchange } A[n - i + 1] \text{ and } A[\text{max}]$ 
```

## Algorithm Running Time Analysis: Example 2 Solution

Algorithm: $\text{alg}(A, n)$		Cost	Number of times executed
1	<b>for</b> $i = 1$ <b>to</b> $\lfloor n / 2 \rfloor$ <b>do</b>	$c_1$	$\lfloor n / 2 \rfloor + 1$
2	$\text{min} = i$	$c_2$	$\lfloor n / 2 \rfloor$
3	$\text{max} = n - i + 1$	$c_3$	$\lfloor n / 2 \rfloor$
4	<b>if</b> $A[\text{min}] > A[\text{max}]$ <b>then</b>	$c_4$	$\lfloor n / 2 \rfloor$
5	exchange $A[\text{min}]$ and $A[\text{max}]$	$c_5$	$\alpha \cdot \lfloor n / 2 \rfloor; \quad 0 \leq \alpha \leq 1$
6	<b>for</b> $j = i + 1$ <b>to</b> $n - i$ <b>do</b>	$c_6$	
7	<b>if</b> $A[j] < A[\text{min}]$ <b>then</b>	$c_7$	
8	$\text{min} = j$	$c_8$	
9	<b>if</b> $A[j] > A[\text{max}]$ <b>then</b>	$c_9$	
10	$\text{max} = j$	$c_{10}$	
11	exchange $A[i]$ and $A[\text{min}]$	$c_{11}$	
12	exchange $A[n - i + 1]$ and $A[\text{max}]$	$c_{12}$	



## Algorithm Running Time Analysis: «Table Method» Example

The following example shows how to analyse the number executions of the body of the inner loop (lines 7 to 10) of the algorithm  $\text{alg}(A, n)$  using a table to track the evolution of the number of executions depending on the value of the outer loop variable.

Outer loop variable value $i =$	Inner loop variable range $j = i+1 \dots n-i$	Number of executions of the inner loop body
1	2 ... $n-1$	$n - 2$
2	3 ... $n-2$	$n - 4$
3	4 ... $n-3$	$n - 6$
$\vdots$	$\vdots$	$\vdots$
k	$k+1 \dots n-k$	$n - 2 \cdot k$
$\vdots$	$\vdots$	$\vdots$
$\lfloor n/2 \rfloor$	$\lfloor n/2 \rfloor + 1 \dots \lfloor n/2 \rfloor$	0

Algorithm:  $\text{alg}(A, n)$

```

1 for  $i = 1$  to  $\lfloor n/2 \rfloor$  do
2    $min = i$ ;
3    $max = n - i + 1$ ;
4   if  $A[min] > A[max]$  then
5     exchange  $A[min]$  and  $A[max]$ ;
6   for  $j = i + 1$  to  $n - i$  do
7     if  $A[j] < A[min]$  then
8        $min = j$ ;
9     if  $A[j] > A[max]$  then
10       $max = j$ ;
11  exchange  $A[i]$  and  $A[min]$ ;
12  exchange  $A[n - i + 1]$  and  $A[max]$ ;

```

## Algorithm Running Time Analysis: «Table Method» Example

The total number of execution within both of the nested loop (body) is the sum of the executions of the inner loop over all iterations of the outer loop:

$$\sum_{k=1}^{\lfloor n/2 \rfloor} n - 2 \cdot k$$

This sum can be resolved as follows:

$$\sum_{k=1}^{\lfloor n/2 \rfloor} n - 2 \cdot k = \sum_{k=1}^{\lfloor n/2 \rfloor} n - \sum_{k=1}^{\lfloor n/2 \rfloor} 2 \cdot k = n \cdot \sum_{k=1}^{\lfloor n/2 \rfloor} 1 - 2 \cdot \sum_{k=1}^{\lfloor n/2 \rfloor} k = n \cdot \lfloor n/2 \rfloor - 2 \cdot \frac{\lfloor n/2 \rfloor \cdot (\lfloor n/2 \rfloor + 1)}{2}$$

## Algorithm Running Time Analysis: «Nested Multiple Sum» Example

$$\sum_{i=1}^{\lfloor n/2 \rfloor} \sum_{j=i+1}^{n-i} 1$$

Algorithm:  $\text{alg}(A, n)$

```
1 for  $i = 1$  to  $\lfloor n/2 \rfloor$  do
2    $min = i$ ;
3    $max = n - i + 1$ ;
4   if  $A[min] > A[max]$  then
5     exchange  $A[min]$  and  $A[max]$ ;
6   for  $j = i + 1$  to  $n - i$  do
7     if  $A[j] < A[min]$  then
8        $min = j$ ;
9     if  $A[j] > A[max]$  then
10       $max = j$ ;
11  exchange  $A[i]$  and  $A[min]$ ;
12  exchange  $A[n - i + 1]$  and  $A[max]$ ;
```

## Exercise 3, Task 1

**Algo:** whatDoesItDo( $A, n, k$ )

---

*result* = -1000

**for**  $i = 1$  **to**  $n$  **do**

*current* = 0

**for**  $j = i$  **to**  $n$  **by**  $k$  **do**

*current* = *current* +  $A[j]$

**if** *current* > *result* **then**

*result* = *current*

**return** *result*

## Tips and Tricks for Finding Out What an Algorithm Does

- Having experience (i.e. having seen, thought about and understood many algorithms) and practicing definitely helps.
- Looking at the variable names may help (assuming they are not intentionally misleading).
- Find out what parts of the code are intended for (i.e. lines 8 to 10 are a swap) and wrap them into an abstraction (i.e. write «swap  $A[i]$  and  $A[\max]$ » instead of lines 8 to 10).
- Make a drawing of the array on which the algorithm operates and think about how the loops go over it.
- Make a table and trace how the variables and their values change.
- Calculate a set of input and output values and compare them.
- ...

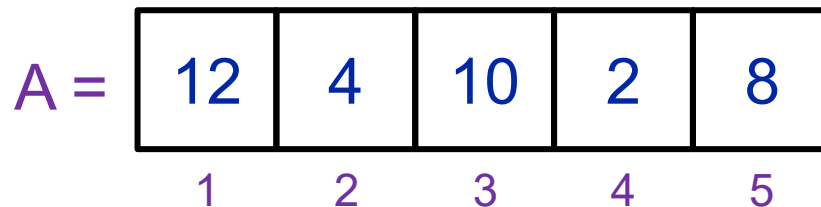
↓ increasing desperation

## Exercise 3, Task 1a and 1b

a) What does algorithm *whatDoIDo*( $A, n, k$ ) do?

This algorithm is an implementation of a descending selection sort with simultaneous summation, restricted to the first  $k$  elements of the input array: The algorithm will sort the first  $k$  elements of the array in descending order and at the same time will calculate and return the sum of the  $k$  biggest elements by adding up the current maximum.

For example, given the input array  $A = [12, 2, 10, 4, 8]$  and  $k = 3$ , the return value will be  $12 + 10 + 8 = 30$  and the state of array  $A$  after the algorithm will be  $A' = [12, 10, 8, 2, 4]$ .



**Algo:** *WHATIDO*( $A, n, k$ )

```
sum = 0;
for  $i = 1$  to  $k$  do
    maxi = i;
    for  $j = i + 1$  to  $n$  do
        if  $A[j] > A[maxi]$  then
            maxi = j;
    sum = sum +  $A[maxi]$ ;
    swp =  $A[i]$ ;
     $A[i] = A[maxi]$ ;
     $A[maxi] = swp$ ;
return sum
```

## Exercise 3, Task 1c and 1d

Instruction	# of times executed	Cost
<b>result</b> = -1000	1	$c_1$
<b>for</b> $i := 1$ <b>to</b> $n$ <b>do</b>	$n + 1$	$c_2$
$current = 0$	$n$	$c_3$
<b>for</b> $j := i$ <b>to</b> $n$ <b>by</b> $k$ <b>do</b>	$\frac{n^2 - n}{2k} + 2n^*$	$c_4$
$current = current + A[j]$	$\frac{n^2 - n}{2k} + n^{**}$	$c_5$
<b>if</b> $current > result$ <b>then</b>	$n$	$c_6$
$result = current$	$\alpha n^{***}$	$c_7$
<b>return</b> $result$	1	$c_8$

## Exercise 3, Task 1e

```
void maximal_sum_every_k(int A[], int n, int k) {  
    int result = -1000;  
    int i;  
    int j;  
    for (i = 0; i < n; i++) {  
        int current = 0;  
        for (j = i; j < n; j += k) {  
            current = current + A[j];  
        }  
        if (current > result) {  
            result = current;  
        }  
    }  
    printf("Result: %d\n", result);  
}
```

**Algo:** whatDoesItDo( $A, n, k$ )

---

```
result = -1000  
for  $i = 1$  to  $n$  do  
     $current = 0$   
    for  $j = i$  to  $n$  by  $k$  do  
         $current = current + A[j]$   
    if  $current > result$  then  
         $result = current$   
return result
```



## Exercise 3, Task 3

**Algo:** algo1( $A, n, k$ )

---

```
sum = 0;
for  $i = 1$  to  $k$  do
     $maxi = i$ ;
    for  $j = i$  to  $n$  do
        if  $A[j] > A[maxi]$  then
             $maxi = j$ ;
     $sum = sum + A[maxi]$ ;
     $swp = A[i]$ ;
     $A[i] = A[maxi]$ ;
     $A[maxi] = swp$ ;
return  $sum$ 
```

## Exercise 3, Task 3a: Solution

The preconditions (inputs) are an array  $A[1..n]$  and an integer  $k$ .

The post conditions(outputs) are the following:

- sum of the biggest  $k$  elements of the array  $A[1..n]$ . Recursively, we can define the output of `algo1` (sum of the biggest  $k$  elements of the array  $A[1..n]$ ) in the following way: Let  $sum \in \mathbb{N}$  denote the biggest  $k$  elements of the array  $A[1..n]$ , then we have

$\forall i \in [1..k] : sum = sum + A[i]$ , where  $A[1..k]$  are the biggest  $k$  integers and sorted in a descending order

- Integers of  $A[1..k]$  are the biggest  $k$  integers in  $A$  and sorted in a descending order.

## Exercise 3, Task 3b: Solution

- i. Determine if the loop is up loop or down loop.

The outer loop `for i = 1 to k` is an up loop, as it runs from low (1) to high  $k$ .

The inner loop `for j = i to n` is an up loop as well, as it runs from low ( $i$ ) to high  $n$ .

- ii. Determine the invariants of these two loops and verify whether they hold in three stages: **initialization**, **maintenance** and **termination**.

We start with the inner loop. The invariant of the inner loop is that  $\forall k \in [i..j] : A[*maxi*] \geq A[k]$ , i.e.,  $A[*maxi*]$  is the largest element in the array  $A[i..j]$ .

- Initialization.** This invariant holds. At this time,  $j = i$  and  $A[i, j]$  contains only one element  $A[i]$ . At this moment (before the loop starts), we have  $j = i$  and  $A[*maxi*]$  is the only, and the biggest element in  $A[i, j]$ .
- Maintenance.** This invariant holds. If  $A[*maxi*] \geq A[m], \forall m \in [i, j]$  (before the loop), and  $A[j] > A[*maxi*]$ , then  $maxi$  is assigned to be  $j$ . In this case, we still have  $A[*maxi*] \geq A[m], \forall m \in [i..j]$  (after the loop).
- Termination.** The inner loop terminates when  $j = n$ . At this time, if  $A[j] = A[i] > A[*maxi*]$ , then  $maxi$  is assigned to be  $n$ . It allows us to conclude that  $A[*maxi*] > A[m], \forall m \in [i..j]$ . Furthermore, we also have  $A[*maxi*] > A[m], \forall m \in [i..n]$  after the loop terminates. In other words,  $A[*maxi*]$  is the biggest element in  $A[i..n]$ .

## Exercise 3, Task 3b: Solution

Then we analyse the outer loop, **for**  $i = 1$  **to**  $k$ . The invariant of the outer loop is that  $A[1..i]$  is sorted in descending order and contains the biggest  $i$  elements of the array  $A[1..n]$ .

- i. **Initialization.** This invariant holds. Before the loop begins,  $i = 1$  and  $A[1..i]$  contains only one element. It is naturally sorted and has the largest element of the subarray  $A[1..1]$ .
- ii. **Maintenance.** This invariant holds. We assume  $A[1..i-1]$  is sorted in descending order and contains the largest  $i - 1$  elements of the array  $A[1..n]$  before the loop. The inner loop guarantees that  $maxi$  is the index to the biggest element in  $A[i..n]$ . Since  $A[1..i-1]$  already contains the biggest  $i - 1$  elements of  $A[1..n]$ , we have

$$\forall p \in [i..n], \forall q \in [1..i-1], A[q] \geq A[p]$$

In other words,  $maxi$  is the index to the biggest element in  $A[i..n]$  but it is still smaller than any element in  $A[1..i-1]$ , which implies it is the  $i$ th biggest element. The body of the outer loop swaps it with the  $i$ th element in  $A[1..n]$ , which keeps  $A[1..i]$  the biggest  $i$  elements of  $A[1..n]$  and sorted in descending order.

- iii. **Termination.** The loop terminates when  $i = k$ . At this time,  $A[1..k]$  is sorted in descending order and contains the largest  $k$  elements of  $A[1..n]$ .

## Exercise 3, Task 3c: Solution

Identify some edge cases of the algorithm and verify if the algorithm has the correct output.

- If  $A[1..n]$  is empty, then the algorithm only initialize `sum` to be zero and returns it.
- If  $A[1..n]$  only contains one element, the outer loop will be executed only once and guarantees the  $A[1..n]$  contains the biggest element, which is the only element in the array. The algorithm returns the initialized sum (0) plus the only element in the array ( $A[1]$ ).
- For a general case, the outer loop guarantees that  $A[1..n]$  contains the biggest  $k$  elements. In the body of the outer loop, the algorithm calculates the sum of the first  $k$  elements in the array  $A[1..n]$  and returns it. The returned value is the sum of the biggest  $k$  elements.

## Exercise 3, Task 3d: Solution

Instruction	# of times executed	Cost
<i>sum</i> := 0	1	$c_1$
<b>for</b> <i>i</i> := 1 <b>to</b> <i>k</i> <b>do</b>	$k + 1$	$c_2$
<i>maxi</i> := <i>i</i>	$k$	$c_3$
<b>for</b> <i>j</i> := <i>i</i> <b>to</b> <i>n</i> <b>do</b>	$\left(kn - \frac{k(k+1)}{2}\right)^* + k^{**}$	$c_4$
<b>if</b> $A[j] > A[\textit{maxi}]$ <b>then</b>	$kn - \frac{k(k+1)}{2}$	$c_5$
<i>maxi</i> := <i>j</i>	$\alpha \left(kn - \frac{k(k+1)}{2}\right)^{***}$	$c_6$
<i>sum</i> := <i>sum</i> + $A[\textit{maxi}]$	$k$	$c_7$
<i>swp</i> := $A[i]$	$k$	$c_8$
$A[i]$ := $A[\textit{maxi}]$	$k$	$c_9$
$A[\textit{maxi}]$ := <i>swp</i>	$k$	$c_{10}$
<b>return</b> <i>sum</i>	1	$c_{11}$

## Exercise 3, Task 3e: Solution

### Best case

In the best case, the array has already been sorted in descending order, hence we do not need to run `maxi := j`, i.e.,  $\alpha = 0$ . In this case,

$$T_{\text{best}}(n) = c_1 + c_2(k+2) + c_3k + c_4\left((k+1)n - \frac{k(k+1)}{2} + k\right) + c_5\left((k+1)n - \frac{k(k+1)}{2}\right) + 0 + (c_7 + c_8 + c_9 + c_{10})k + c_{11}$$

$$T_{\text{best}}(n) = O(k * n)$$

### Worst case

Similarly, in the worst case, the array is sorted in ascending order and we have to update `maxi` every time, i.e.,  $\alpha = 1$ . In this case,

$$T_{\text{worst}}(n) = c_1 + c_2(k+2) + c_3k + c_4\left((k+1)n - \frac{k(k+1)}{2} + k\right) + c_5\left((k+1)n - \frac{k(k+1)}{2}\right) + c_6\left((k+1)n - \frac{k(k+1)}{2}\right) + (c_7 + c_8 + c_9 + c_{10})k + c_{11}$$

$$T_{\text{worst}}(n) = O(k * n)$$

### Asymptotic complexity of best and worst case

$$T_{\text{best}}(n) = O(k * n)$$

$$T_{\text{worst}}(n) = O(k * n)$$



## Exercise 3 – Task 3e: Remark

*Comprehension question:*

In the lecture slides (SL02/45), it is said, that  $\Omega$  notation (non-tight, non-strict lower bound) is used for best case analysis. Shouldn't we therefore use  $\Omega$  notation here for the best case?





# Algorithm Running Time Analysis: Summation Rules

- General Rules and «Non-Rules»
- Arithmetic Series
- Series of Polynomial Expressions
- Geometric Series
- Harmonic Series
- Arithmetico-Geometric Series
- Decomposing Sums
- Changing Summation Indexes
- Double Sums / Multiple Sums / Nested Sums

## Summations: General Rules / Simple Manipulations

- Summation of constants are equivalent to multiplication (take care about the number of summands):

$$\sum_{k=1}^n c = c + c + \dots + c = n \cdot c \quad (\text{for any constant } c)$$

- Constant multiplication factors in summations can be «pulled out» of the summations:

$$\sum_{k=1}^n c \cdot a_k = c \cdot a_1 + c \cdot a_2 + \dots + c \cdot a_n = c \cdot \sum_{k=1}^n a_k \quad (\text{for any constant } c)$$

- Sums in summations can be split up into separate summations (with both the same indexing):

$$\sum_{k=1}^n a_k + b_k = (a_1 + b_1) + (a_2 + b_2) + \dots + (a_n + b_n) = \sum_{k=1}^n a_k + \sum_{k=1}^n b_k$$

## Summations: «Non-Rules»

- Multiplication:

$$\sum_{k=1}^n (a_k \cdot b_k) \neq \left( \sum_{k=1}^n a_k \right) \cdot \left( \sum_{k=1}^n b_k \right)$$

- Division:

$$\sum_{k=1}^n (a_k / b_k) \neq \frac{\left( \sum_{k=1}^n a_k \right)}{\left( \sum_{k=1}^n b_k \right)}$$

## Summations: Arithmetic Series

A summation of the form

$$\sum_{k=0}^n a_0 + k \cdot d$$

is called an (finite) arithmetic series with  $d = a_{k+1} - a_k = \text{const.}$

The finite arithmetic series sums up to  $(n + 1) \cdot \frac{a_0 + a_n}{2}$  where  $a_n = a_0 + n \cdot d$



## Summations: Arithmetic Series

There is one particularly famous and important (and often used) case of the arithmetic series:

$$\sum_{k=0}^n k = \sum_{k=1}^n k = \frac{n \cdot (n + 1)}{2}$$

(Gauß'sche Summenformel,  $a_0 = 0$ ,  $d = 1$ )

## Summations: Polynomial Expressions

- Sum of squares:

$$\sum_{k=1}^n k^2 = 1^2 + 2^2 + 3^2 + \dots + n^2 = \frac{n \cdot (n+1) \cdot (2n+1)}{6} = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6} \in \Theta(n^3)$$

- Sum of cubes:

$$\sum_{k=1}^n k^3 = 1^3 + 2^3 + 3^3 + \dots + n^3 = \left( \sum_{k=1}^n k \right)^2 = \left( \frac{n \cdot (n+1)}{2} \right)^2 = \frac{n^4}{4} + \frac{n^3}{2} + \frac{n^2}{4} \in \Theta(n^4)$$

- Sum of fourths:

$$\sum_{k=1}^n k^4 = 1^4 + 2^4 + 3^4 + \dots + n^4 = \frac{n \cdot (n+1) \cdot (2n+1) \cdot (3n^2 + 3n - 1)}{30} = \frac{n^5}{5} + \frac{n^4}{2} + \frac{n^3}{3} + \frac{n}{30} \in \Theta(n^5)$$



## Summations: Polynomial Expressions

- Sum of the square of a sum:

$$\sum_{k=1}^n (a_k + b_k)^2 = (a_1 + b_1)^2 + (a_2 + b_2)^2 + \dots + (a_n + b_n)^2 = \sum_{k=1}^n a_k^2 + 2 \cdot \sum_{k=1}^n a_k \cdot b_k + \sum_{k=1}^n b_k^2$$

## Summations: Geometric Series

A summation of the form

$$\sum_{k=0}^n a^k = 1 + a^1 + a^2 + a^3 + \dots + a^n$$

where  $a$  is some real number with  $0 < a \neq 1$  and  $n$  is an integer with  $n > 0$  is called a (finite) geometric series.

This finite geometric series sums up to

$$\frac{a^{n+1} - 1}{a - 1} = \frac{1 - a^{n+1}}{1 - a}$$

(If  $a = 0$ , then the series sums just up to  $n + 1$ .)

The infinite geometric series  $\sum_{k=1}^{\infty} a^k = 1 + a^1 + a^2 + a^3 + \dots$

converges to  $1 / (1 - a)$  iff  $|a| < 1$ , otherwise it diverges.



## Summations: Harmonic Series

A summation of the form

$$\sum_{k=1}^n \frac{1}{k} = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n}$$

is called an (finite) harmonic series.

There is no closed formula for these sums.

Note that, although the associated sequence is a zero sequence (Nullfolge; thus its terms come always closer to zero), the *infinite* harmonic series is *divergent*.

## Summations: Arithmetico-Geometric Series

By combining the arithmetic series and the geometric series by a term-by-term multiplication, we get a summation of the form

$$\sum_{k=0}^n (a_0 + k \cdot d) \cdot b^k = a_0 + (a_0 + d) \cdot b + (a_0 + 2 \cdot d) \cdot b^2 + (a_0 + 3 \cdot d) \cdot b^3 + \dots + (a_0 + n \cdot d) \cdot b^n$$

(with  $0 < b \neq 1$ ,  $d = \text{const}$  and an integer  $n > 0$ ).

This is called a (finite) **arithmetico-geometric series**. As it is written above (i.e. starting at  $k = 0$ ) sums up to

$$\frac{a_0}{1-b} - \frac{(a_0 + (n-1) \cdot d) \cdot b^n}{1-b} + \frac{d \cdot b \cdot (1-b^{n-1})}{(1-b)^2}$$

The infinite arithmetico-geometric series sums up to:

$$\sum_{k=0}^{\infty} (a_0 + k \cdot d) \cdot b^k = \frac{a_0}{1-b} + \frac{d \cdot b}{(1-b)^2}$$

## Example: C Code Running Time Analysis with Series

In which asymptotic complexity class is the following C code function?

Write the number of executions of line 6 as a summation formula.

What will be the return value of functionK(100)?

```
int functionK(int n) {  
    int z = 0;  
    for (int i = 1; i <= n; i++) {  
        for (int j = 1; j <= n; j++) {  
            for (int k = 1; k <= j; k++) {  
                z = z + 1;  
            }  
        }  
    }  
    return z;  
}
```

→  $O(n^3)$

$$\begin{aligned} \sum_{i=1}^n \sum_{j=0}^n \sum_{k=1}^j 1 &= \\ &= \sum_{i=1}^n \sum_{j=0}^n j = \sum_{i=1}^n \frac{n \cdot (n+1)}{2} = \\ &= n \cdot \frac{n \cdot (n+1)}{2} = \frac{1}{2} \cdot (n^3 + n^2) = f_K(n) \end{aligned}$$

→  $f_K(100) = 505000$

## Summations: Decomposing Sums

Indexes of sums can be rewritten / sums can be «taken apart» as in the following example:

$$\sum_{k=x}^{n-x} k = \sum_{k=1}^{n-x} k - \sum_{k=1}^{x-1} k$$

↑

sum up «too much», i.e.  
start summing up from 1  
instead of starting from x

↑

remove what has been  
summed up too much, i.e.  
numbers from 1 to x - 1

This is useful in solving sums stemming from nested loops with interacting loop variables.



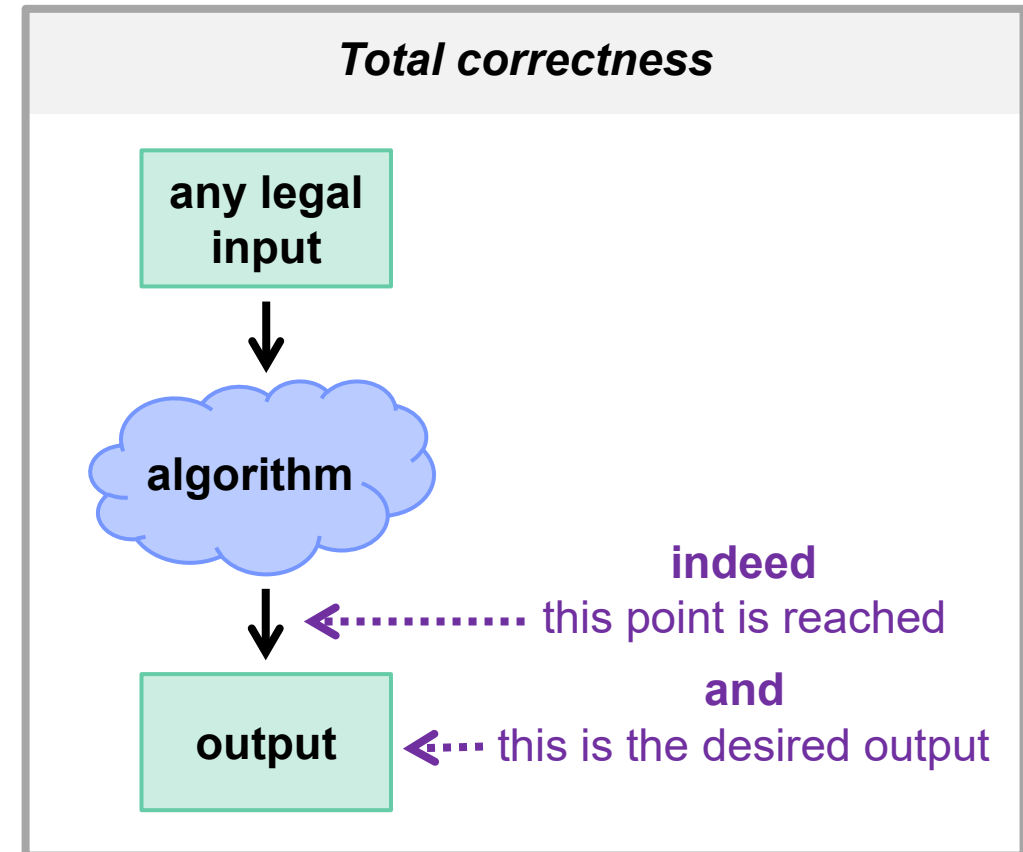
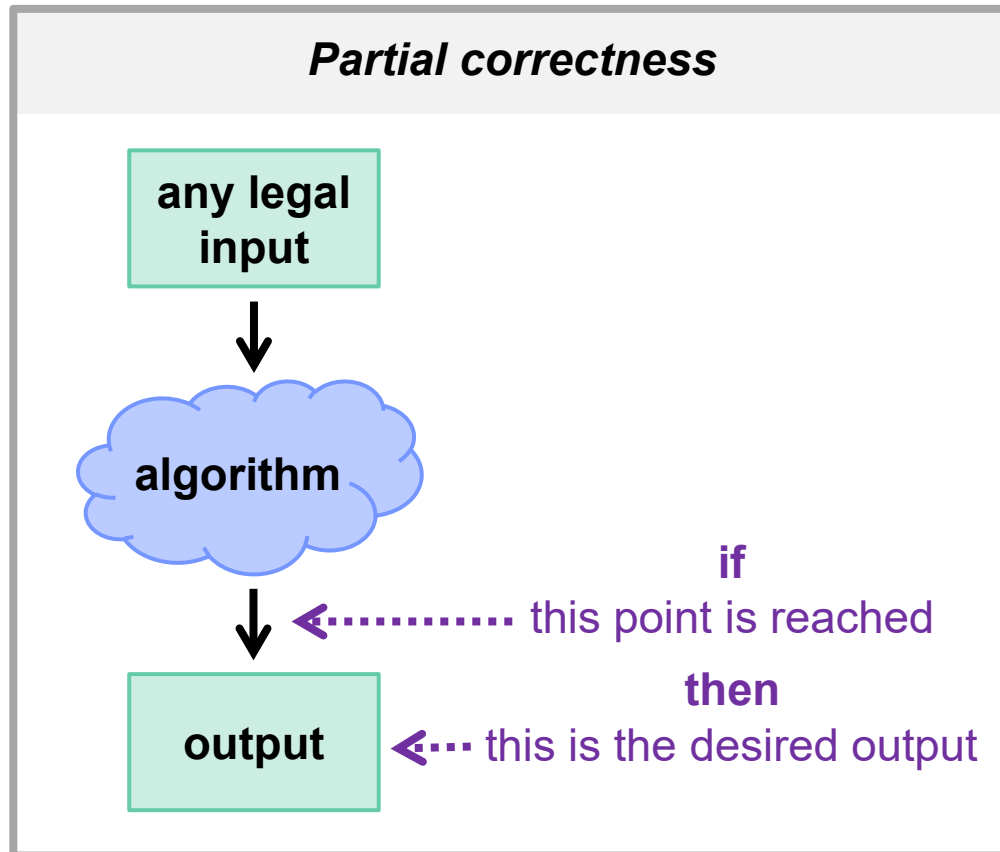
**Universität  
Zürich** UZH

**Institut für Informatik**

# Correctness, Loop Invariants

- Correctness
- Loop Invariants

## Correctness



## Correctness: Example

The adjacent C code is a very simple example of the implementation of an algorithm which is only partially correct but not (totally) correct:

If the input is even, it will return the threefold of the input. If the input is odd, the function will go into an infinite loop and never terminate.

```
1  /**
2   * returns the threefold of the input
3   */
4  int triple(int input) {
5      if (input % 2 == 0) {
6          return input * 3;
7      }
8      else {
9          while (1 == 1) {
10             input = input + 1;
11         }
12     }
13 }
```

## Invariants

Invariants are an **useful tool** for

- **proofing the correctness** of an algorithm and in particular of loops (loop invariants),
- better understanding and reasoning about algorithms in general.

An invariant is a **logical expression**, i.e. a statement which can be evaluated to true or false. This logical expression refers to some property of an algorithm or a part of an algorithm and **has to be true** in a certain range of the control flow for the algorithm to work as it should. Oftentimes, the invariant **captures the main idea** underlying an algorithm.

An invariant could look as follows for **example**:

- «the value of variable  $x$  has always to be positive», formally:  $x > 0$
- «all numbers from 1 to 41 are present somewhere in array  $A$  of length  $n$ »,  
formally:  $\forall x: 0 < x < 42, \exists i \leq n: A[i] = x$



## Loop Invariants

For every loop, a loop invariant can be assigned.

To show that a loop invariant holds, three conditions have to be checked:

- **initialization**: the invariant has to be true (right) before the first iteration of the loop starts
- **maintenance**: if it is true before an iteration then it is true after that iteration; the invariant has to be true right at the start and right at the end of each iteration of the loop
- **termination**: the invariant has to be true (right) after the loop has ended; this is a useful property that helps to show that the algorithm is correct

This technique is similar to an inductive proof.

## Invariant Finding Example

Find an **invariant** for the loop in the following C code fragment:

```
int c = 42;
int x = c;
int y = 0;
while (x > 0)
{
    x = x - 1;
    y = y + 1;
}
```

$c = x + y = 42$

is an invariant of the while loop in this C code fragment.

Note that this is not the only invariant of the loop but there are many possibilities, e.g.

- $y \geq 0$
- $x \geq 0$
- $(x \% 2 = y \% 2) \wedge (x \cdot y \neq 1)$
- ...

Obviously, in practice we will only be interested in an invariant which is useful for understanding the algorithm and proving a certain property of it. What is a helpful invariant, depends on what the algorithm does.

## Loop Invariants: Strategies

Two parts can be distinguished when working with loop invariants:

- finding the loop invariant,
- writing the loop invariant using formal language.

An **unavoidable prerequisite** for formulating a loop invariant is to **understand what the loop actually does** and should accomplish within the algorithm it is part of. Loop invariants are a formalization of one's intuition about how the loop works. For this part, it is difficult to give some general advice or recipes because algorithms use quite different ideas. Some algorithms use make use of similar ideas and it therefore helps if one has seen some examples of loop invariants.

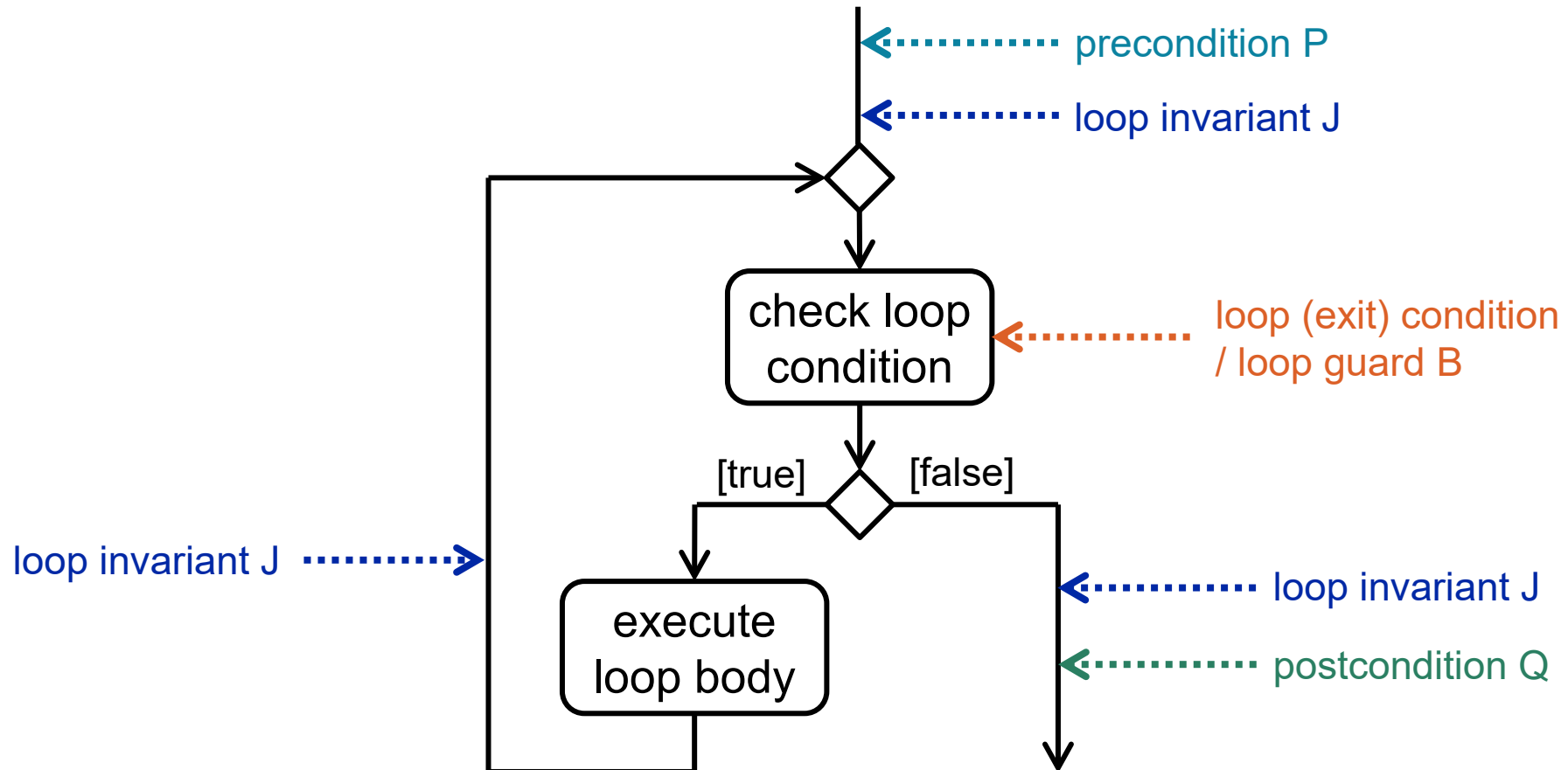
If the loop invariant was found and formulated using natural language, it has to be written using formal language (predicate logic). This part is mainly a matter of training.

## Precondition and Postcondition

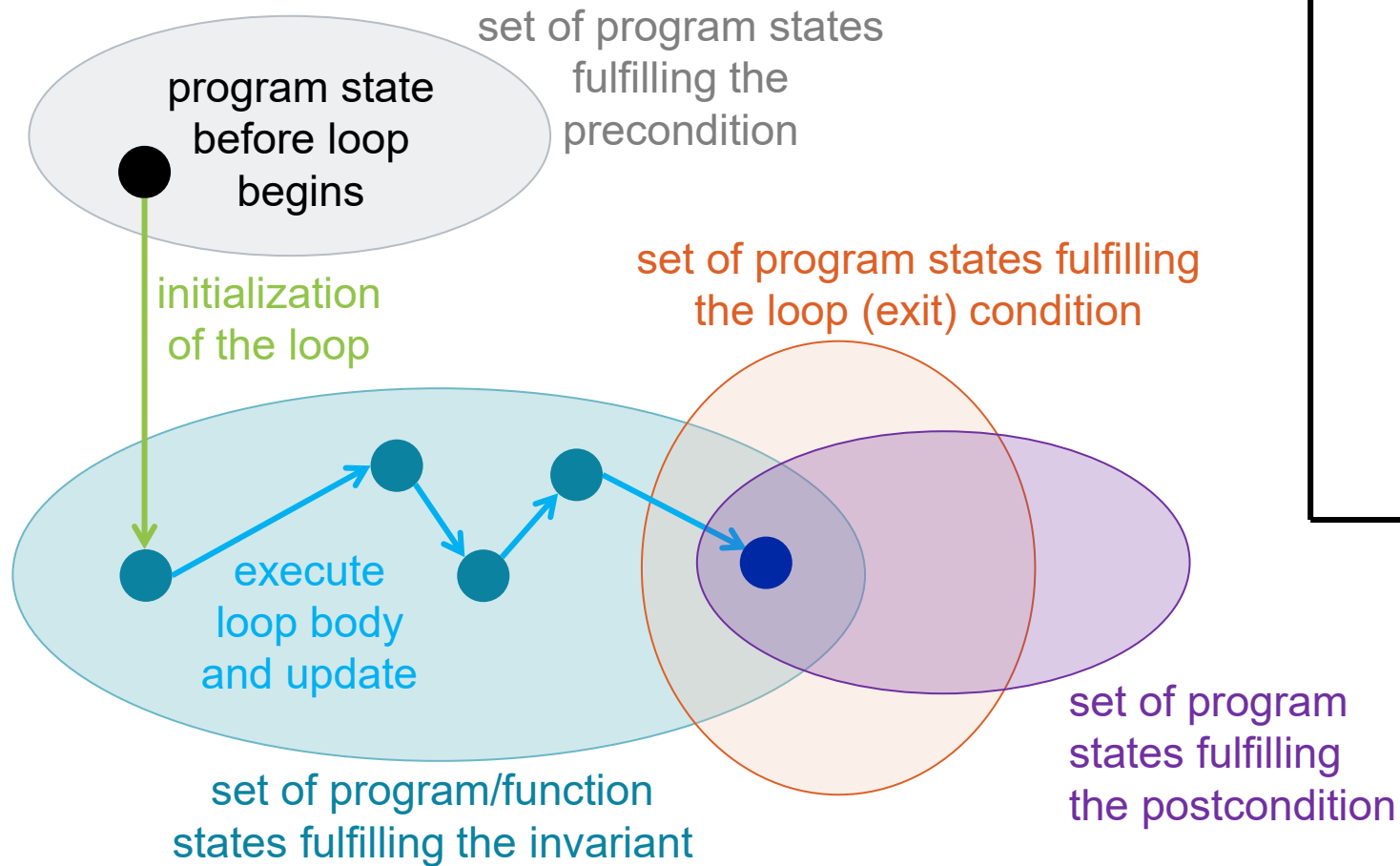
Precondition and postcondition are logical expressions which can be evaluated to true or false.

- A **precondition** defines the range / set of arguments and environment situations which allow a meaningful execution a program, function or part of it. If the input arguments do not fulfill the precondition, the result of the respective calculations are undefined. The responsibility is outsourced to the person calling the respective program function. A precondition could be for example that an input parameter might not be equal to zero (because it is used in a division).
- A **postcondition** is a description of the output or state which is reached after successful execution of a program, function or part of it.

## Loop Invariants, Precondition, Postcondition



## Loop Invariants: Visualization as States and Sets

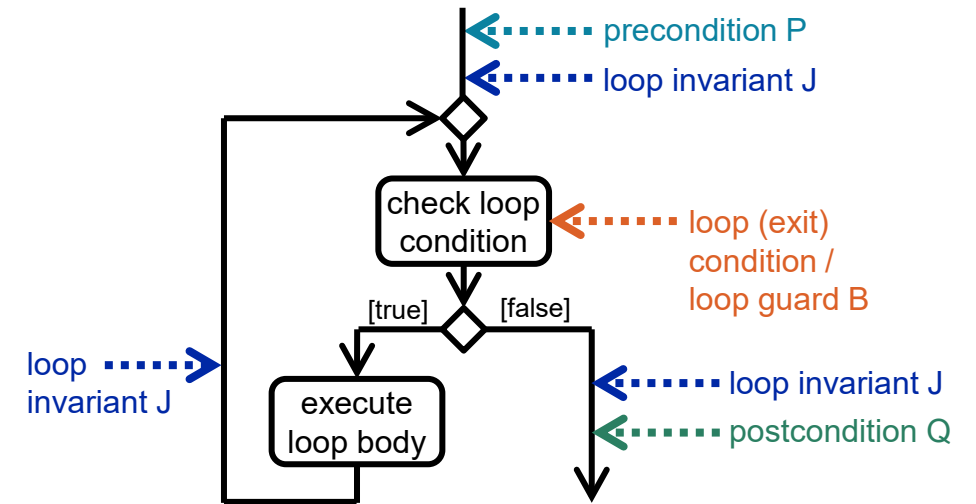


*Example:*

```
int z = 3
for (int i = 0; i < 4; i++)
{
    printf("%d"; i / z);
    //do something
}
```

## Loop Invariants: Formal Description

Using precondition  $P$ , postcondition  $Q$ , loop exit condition  $B$  and loop invariant  $J$ , we can now proceed to a more formal description of the terms initialization, maintenance and termination:



- **Initialization**: Invariant must hold initially:  $P \Rightarrow J$
- **Maintenance**: loop body must re-establish the invariant:  $(J \wedge B) \Rightarrow J$
- **Termination**: Invariant must establish postcondition if loop condition is false:  $(J \wedge \neg B) \Rightarrow Q$

## Expressing Statements in Predicate Logic: Exercise

When dealing with correctness and loop invariants, one difficulty is to be able to precisely express statements about an algorithm mathematically (using predicate logic with quantors  $\exists$  and  $\forall$  and logical operators  $\wedge$  and  $\vee$ ).

Let  $A[1..n]$  be an array of  $n$  integers.

Express the following statements as formulas in predicate logic:

- a) All elements of  $A$  are greater than 42.
- b) All elements with array index greater than 5 are odd.
- c) There are no two elements with identical values in index range from  $a$  (inclusive) to  $b$  (exclusive).
- d) The elements with array index smaller than  $m$  are sorted descendingly.





**Universität  
Zürich** <sup>UZH</sup>

**Institut für Informatik**

# Preview on Exercise 4



**Universität  
Zürich** <sup>UZH</sup>

**Institut für Informatik**

# Wrap-Up

- Summary
- Feedback
- Outlook
- Questions



## Wrap-Up

- Summary



## Outlook on Next Thursday's Lab Session

*Next tutorial:* Wednesday, 23.03.2022, 14.00 h, BIN 0.B.06

*Topics:*

- Review of Exercise 4
- Preview to Exercise 5
- Recurrences (Repeated Substitution, Recursion Tree, Master Method)
- Divide and Conquer
- ...
- ... (your wishes)



**Universität  
Zürich** <sup>UZH</sup>

**Institut für Informatik**

---

**Questions?**



*Thank you for your attention.*