Universität Zürich^UZH

Institut für Informatik

Tutorial Session 3

Friday, 6th of March 2020

Discussion of Exercise 2, Preview on Exercise 3

Asymptotic Complexity, Running Time Analysis

14.15 – 15.45

Y35-F-32

Agenda

- Discussion of Exercise 2
- Asymptotic Complexity
- Algorithm Running Time Analysis
- Preview on Exercise 3

Review of Exercise 2

- Task 1: Recursive Collatz Series
- Task 2: Recursively Producing the Set of Binary Strings Without Consecutive 1s
- Task 3: T-Square Fractal

Exercise 2 – Task 1: Collatz Problem (Hailstone Sequence)

Consider a finite sequence $s(n) = a_1, a_2, \ldots, a_m$ with $a_1 = n$ and $a_m = 1$ that is defined as follows:

$$a_{i+1} = \begin{cases} a_i / 2 & \text{if } a_i \text{ is even} \\ 3a_i + 1 & \text{if } a_i \text{ is odd} \end{cases}$$

1. Determine s(3). s(3) = 3, 10, 5, 16, 8, 4, 2, 1
2. Write a C program with a *recursive* function `void sequence(int n)` that computes and prints all elements of s(n).

Exercise 2 – Task 1: Collatz Problem (Hailstone Sequence)

```c
void sequence(int n) {
   printf("%d ", n);

   if (n == 1) {
      return;
   }

   if (n % 2 == 0) {
      n = n / 2;
   } else {
      n = 3 * n + 1;
   }
   sequence(n);
}
```

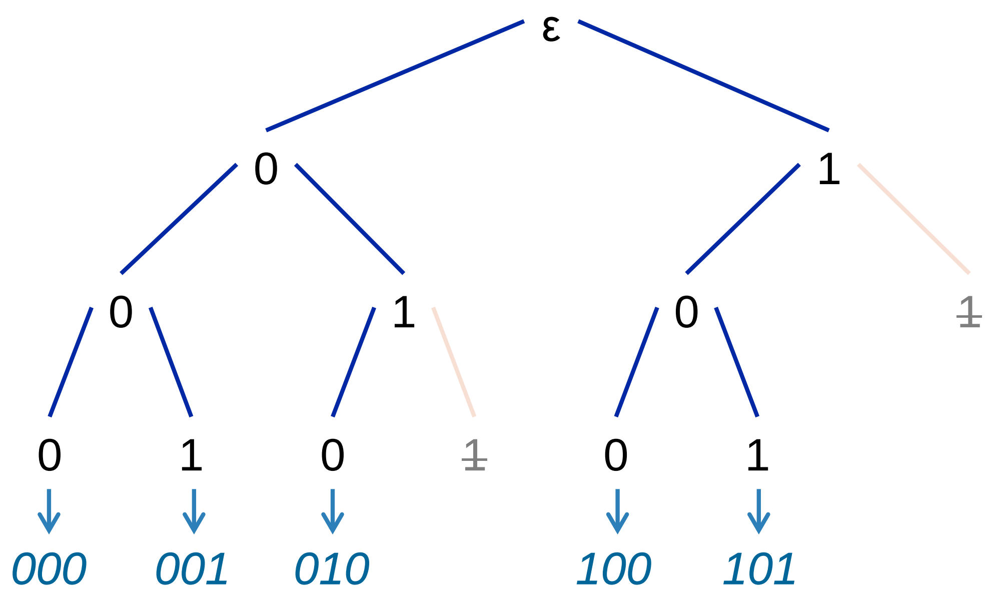
Exercise 2 – Task 2: Set of Binary Strings Without Consecutive 1s

Let B_k be the set of binary strings of length k that do not include consecutive 1s.

1. Determine B_3.
2. Write a C program with a *recursive* function that reads length k from the command line and prints B_k.

Exercise 2 – Task 2: Set of Binary Strings Without Consecutive 1s

Main idea of the algorithm in iterative formulation: Starting from the empty string (ε), we append either a zero or a one to already existing string, although we only append a one if the last character added was a zero (namely to avoid two consecutive ones).



Exercise 2 – Task 2: Set of Binary Strings Without Consecutive 1s: Sample Solution

```c
void generate(char str[], int K, int n) {
    if (n == K) {
        str[n] = '\0';
        printf("%s ", str);
        return;
    }

    if (str[n-1] == '1') {
        str[n] = '0';
        generate(str, K, n+1);
    }
    if (str[n-1] == '0') {
        str[n] = '0';
        generate(str, K, n+1);
        str[n] = '1';
        generate(str, K, n+1);
    }
}
```

off-by-one error

Base case
append null terminator and print

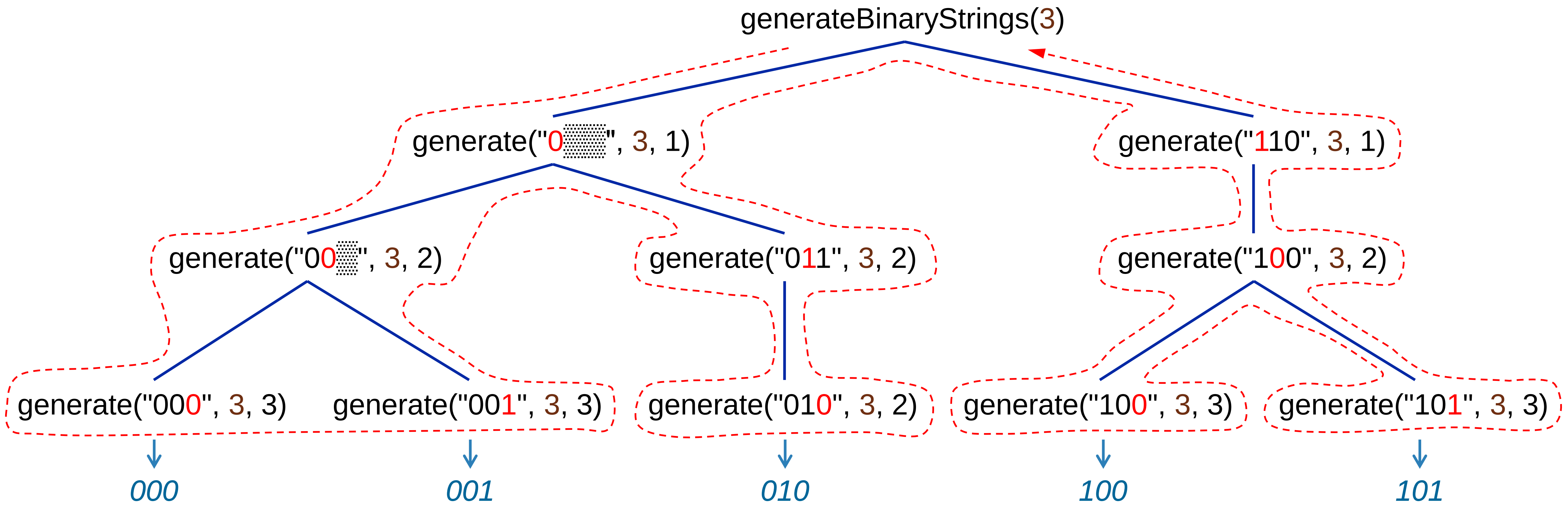
if last character was a one

set current character to '0' and continue recursively

```c
void generateBinaryStrings(int K) {
    if (K <= 0) {
        return;
    }

    char str[K];

    // Generate all binary strings starting with '0'
    str[0] = '0';
    generate(str, K, 1);

    // Generate all binary strings starting with '1'
    str[0] = '1' ;
    generate(str, K, 1);
}
```

should be K+1

Exercise 2 – Task 2: Set of Binary Strings Without Consecutive 1s: Sample Solution Visualized

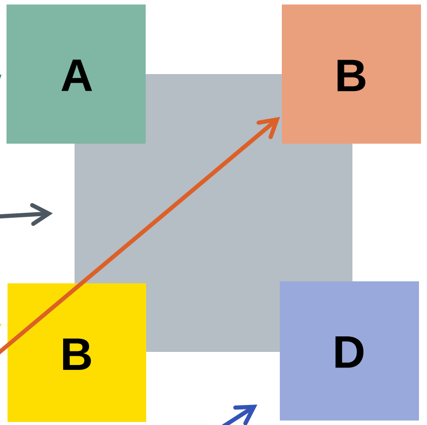


Additional Exercise

- Print recursively the set of all binary strings which do not have three consecutive ones.
- Write in C a recursive function which checks whether a given binary string has three consecutive ones.

Exercise 2 – Task 3: T-Square Fractal

```
void TSquareFractal(double x, double y, double length, int iter) {
    if (iter == 0) {
        return;
    }
    drawsquare(x,y,length);
    SDL_Delay(50);
    TSquareFractal(x-length/4,y-length/4,length/2,iter-1);
    TSquareFractal(x-length/4,y+(3.0/4.0)*length,length/2,iter-1);
    TSquareFractal(x+(3.0/4.0)*length,y-length/4,length/2,iter-1);
    TSquareFractal(x+(3.0/4.0)*length,y+(3.0/4.0)*length,length/2,iter-1);
}
```

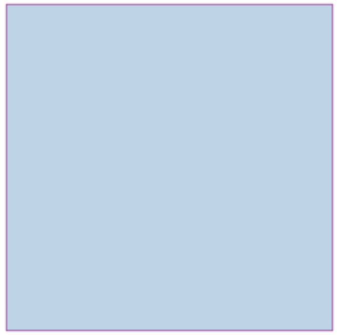


Corner coordinates +/- half of the rescaled size of the square

scaling factor

iteration counter variable

Exercise 2 – Task 3: T-Square Fractal

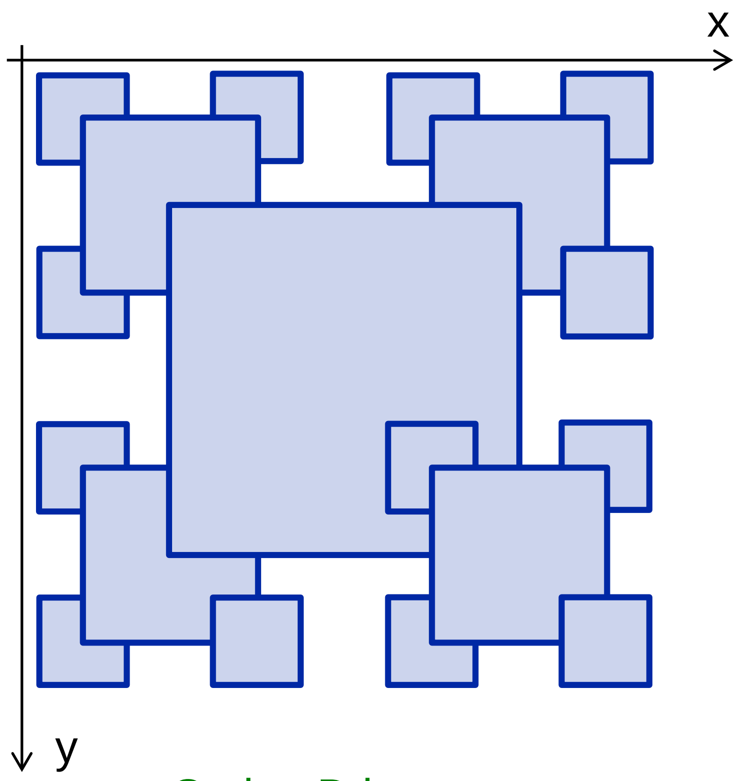


(animation of algorithm; https://www.physik.uzh.ch/~vogelch/algodat/square_fractal_animation.gif)

Exercise 2 – Task 3: T-Square Fractal: Additional Questions

At which position has the call of the drawing function (i.e. `drawsquare(x,y,length);`) to be inserted in order to produce the adjacent image?

```
void TSquareFractal(double x, double y, double length, int iter) {
    if (iter > 0) {
        /* Option A */
        TSquareFractal(x-length/4,y-length/4,length/2,iter-1);
        /* Option B */
        TSquareFractal(x-length/4,y+(3.0/4.0)*length,length/2,iter-1);
        /* Option C */
        TSquareFractal(x+(3.0/4.0)*length,y-length/4,length/2,iter-1);
        /* Option D */            D
        TSquareFractal(x+(3.0/4.0)*length,y+(3.0/4.0)*length,length/2,iter-1);
        /* Option E */
    }
    else {
        return;
    }
}
```



Option D is correct.

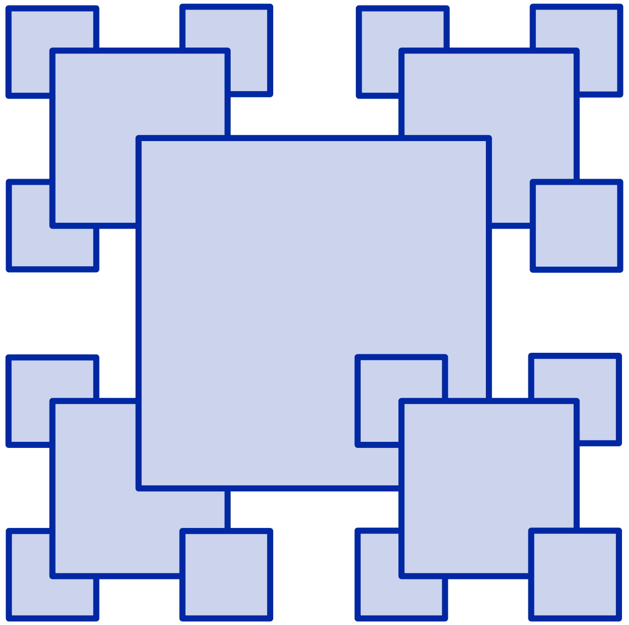
Exercise 2 – Task 3: T-Square Fractal: Additional Questions

What will be the size of the smallest squares in the image, assuming the initial square had a size of 4096 px × 4096 px?

reduced by 2^10 -> 4 pixels

How many squares will be drawn for 10 iterations?

4^0.. + 4^10



Additional Exercise: Fractals and Recursion

Universität Zürich^UZH

Institut für Informatik

Asymptotic Complexity, Landau Notation

- Asymptotic Complexity Illustrated
- 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

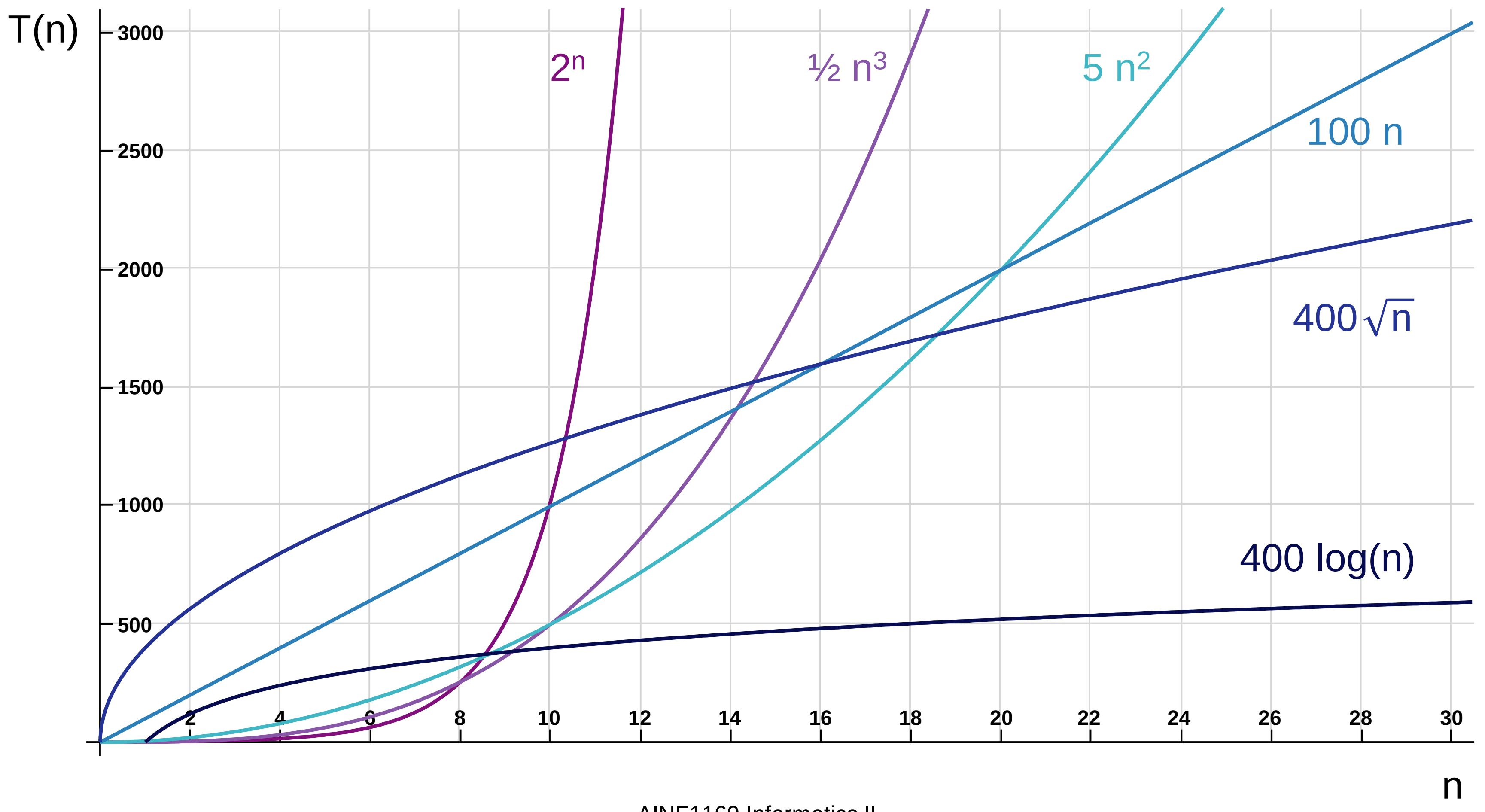
$$f(n) \leq c \cdot g(n) \quad \forall n \geq n_0$$

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

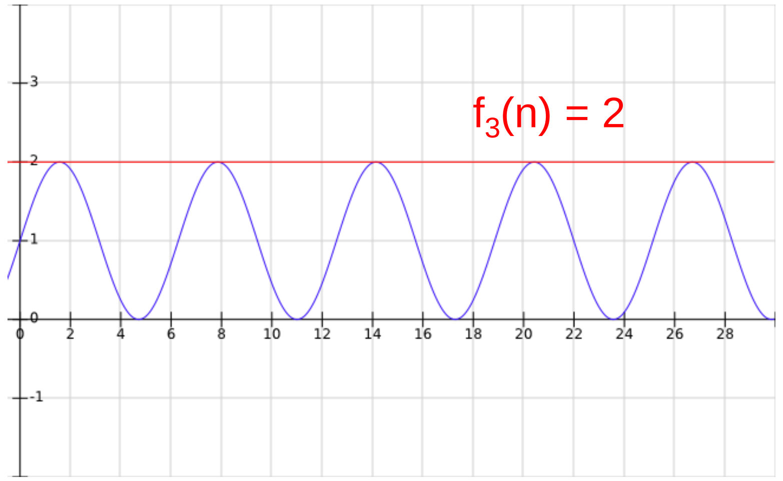
Asymptotic Complexity Illustrated



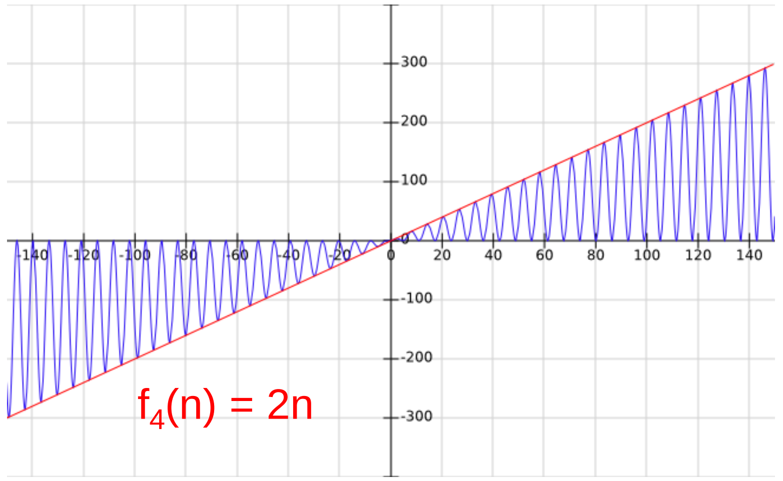
Asymptotic Complexity Example

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

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



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



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 explicitely 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 $n^{½}$ 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

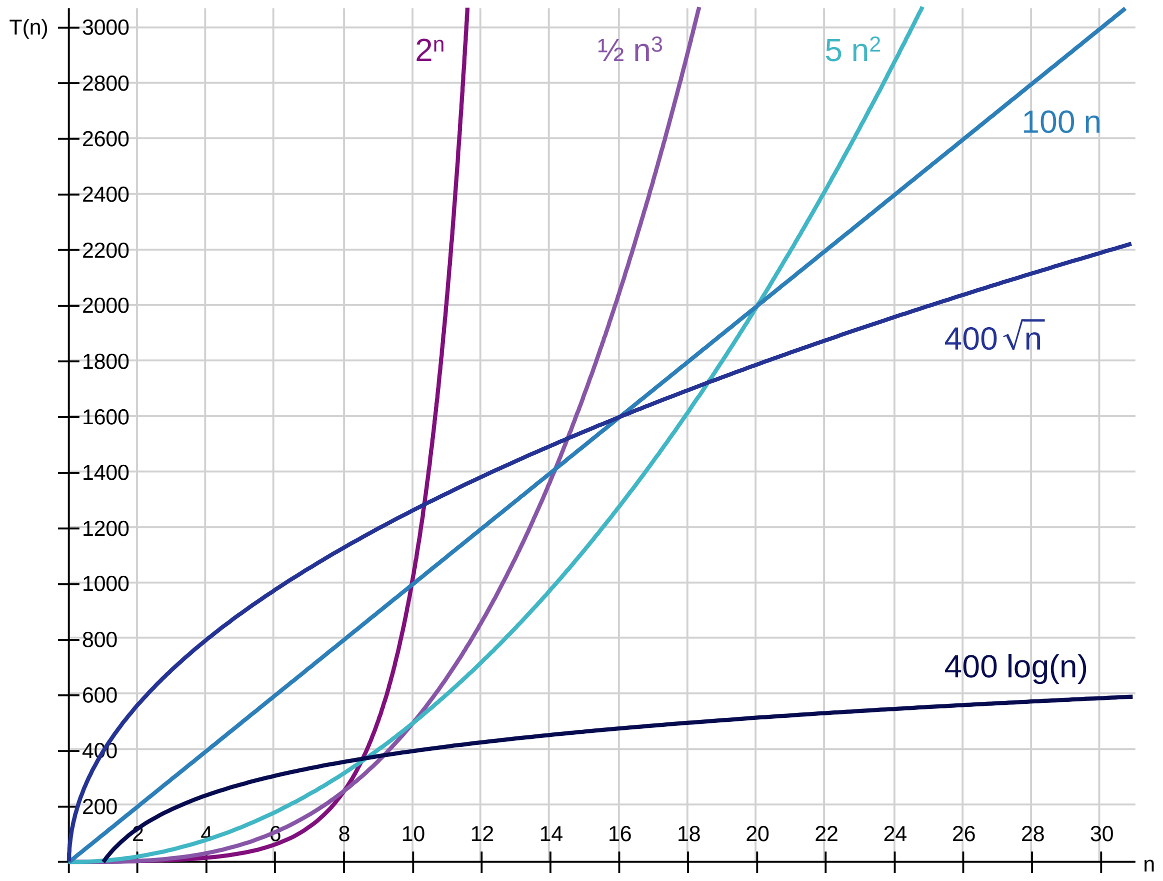
$\Theta(n^{½})$

grows faster than

$\Theta(\log(n))$

grows faster than

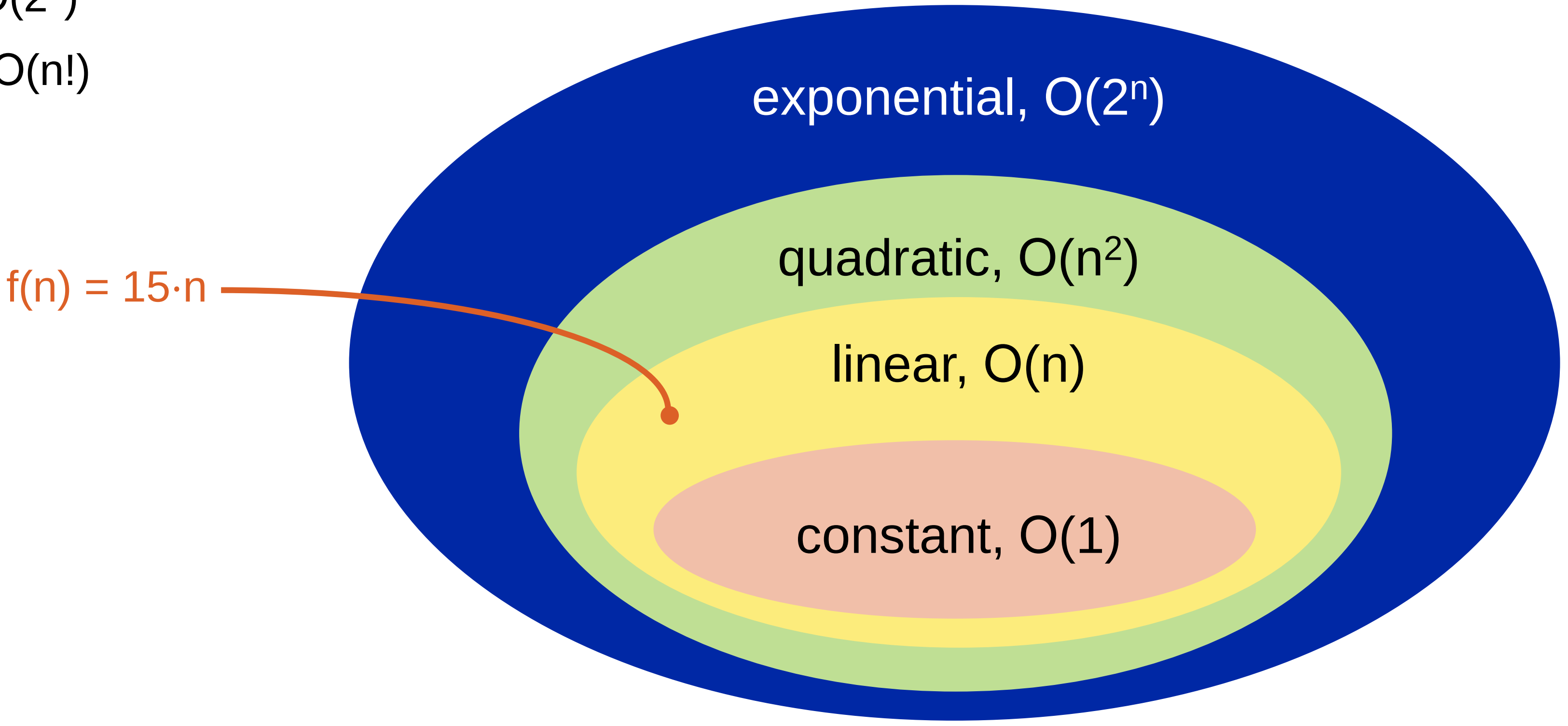
$\Theta(1)$



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.:

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



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): $a^{\log_b(x)} \overset{(4)}{=} \left(b^{\log_b(a)}\right)^{\log_b(x)} = b^{\log_b(a) \cdot \log_b(x)} = b^{\log_b(x) \cdot \log_b(a)} = x^{\log_b(a)}$

Rules for and Some Notes on Asymptotic Complexity

Any addition of constants can be ignored:

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

iff c = const

Example: n^2 is in $O(n^2)$ and $n^2 + 100000000000$ is in $O(n^2)$

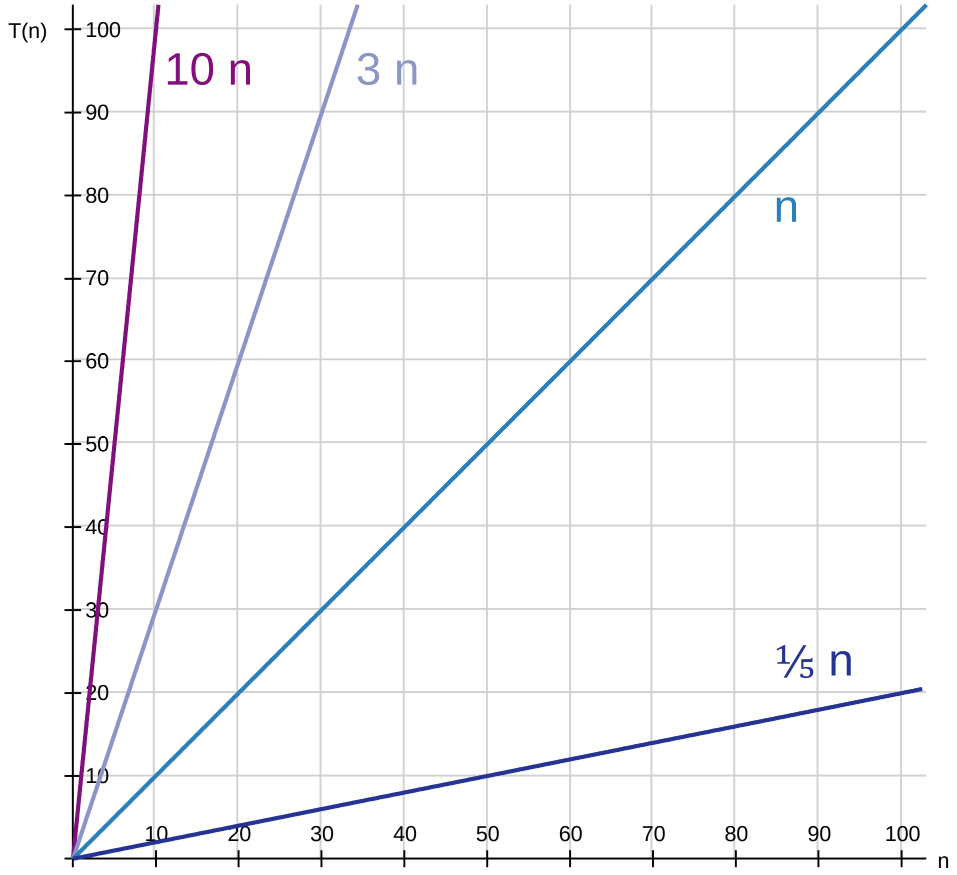
Remark: In coding practice there are some constant additional factors which might not always be ignored, for example writing and reading from disk is usually very slow and might outweigh even complexity class unless for very large data sets.

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)

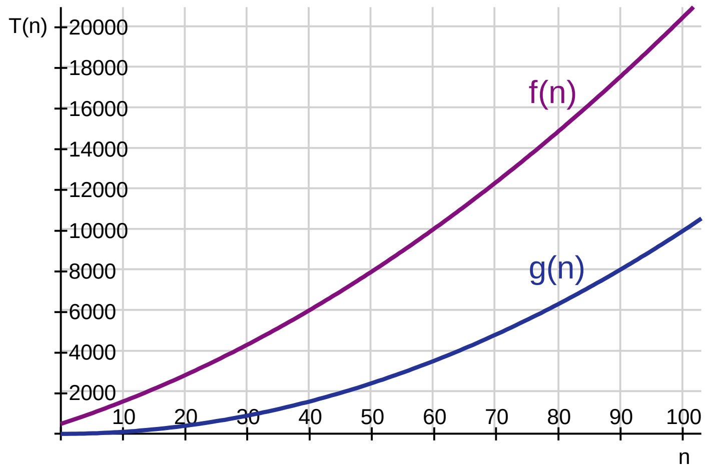


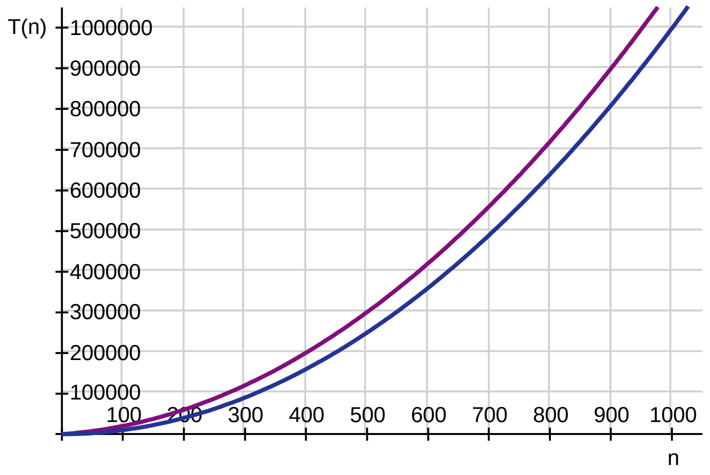
Rules for and Some Notes on Asymptotic Complexity: Sums

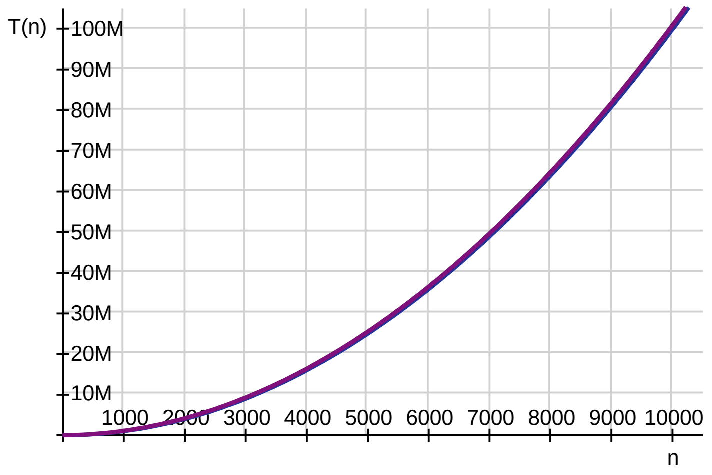
Lower order terms get irrelevant asymptotically. In an sum of functions, the fastest growing function wins:

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

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



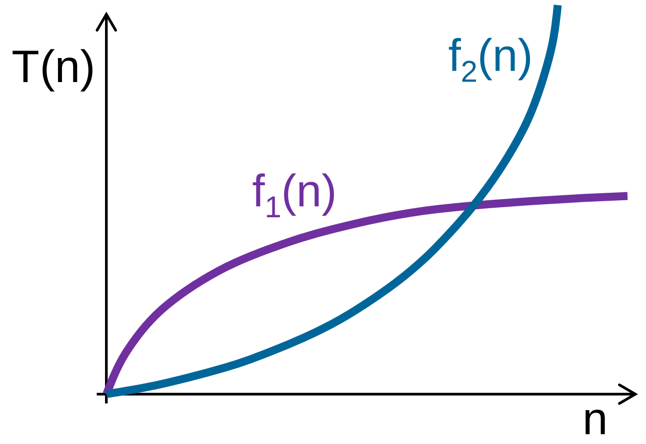


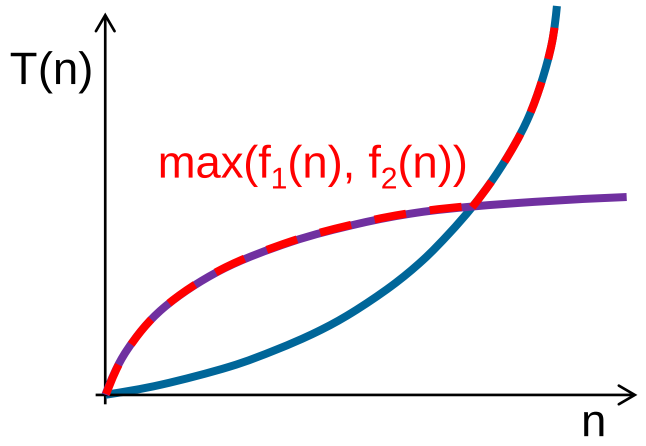


Rules for and Some Notes on Asymptotic Complexity: Sums

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

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





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

Rules for and Some Notes on Asymptotic Complexity: Products

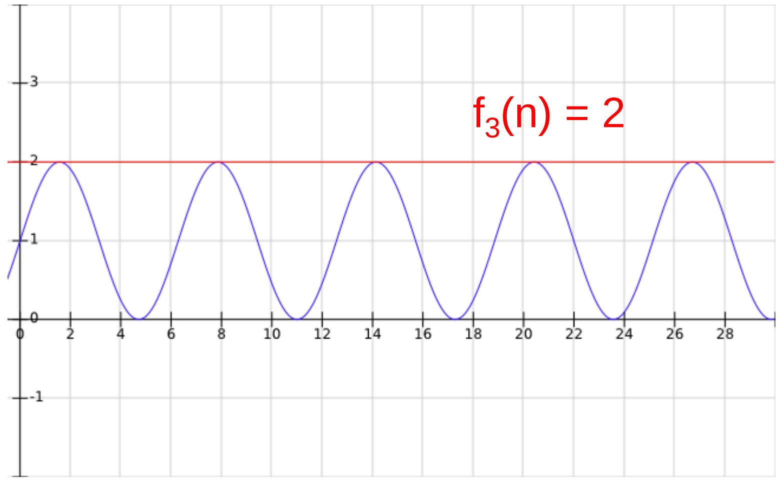
The asymptotic complexity of the product of two functions is:

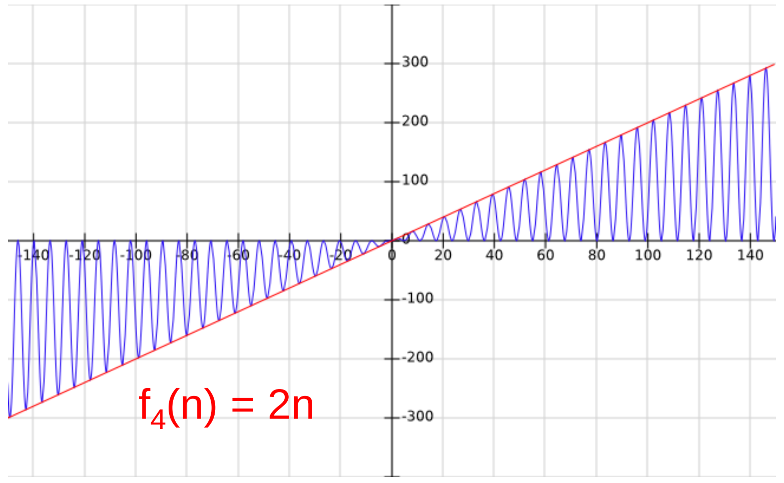
$$(f(n) \cdot g(n)) \in O(f(n) \cdot g(n))$$

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

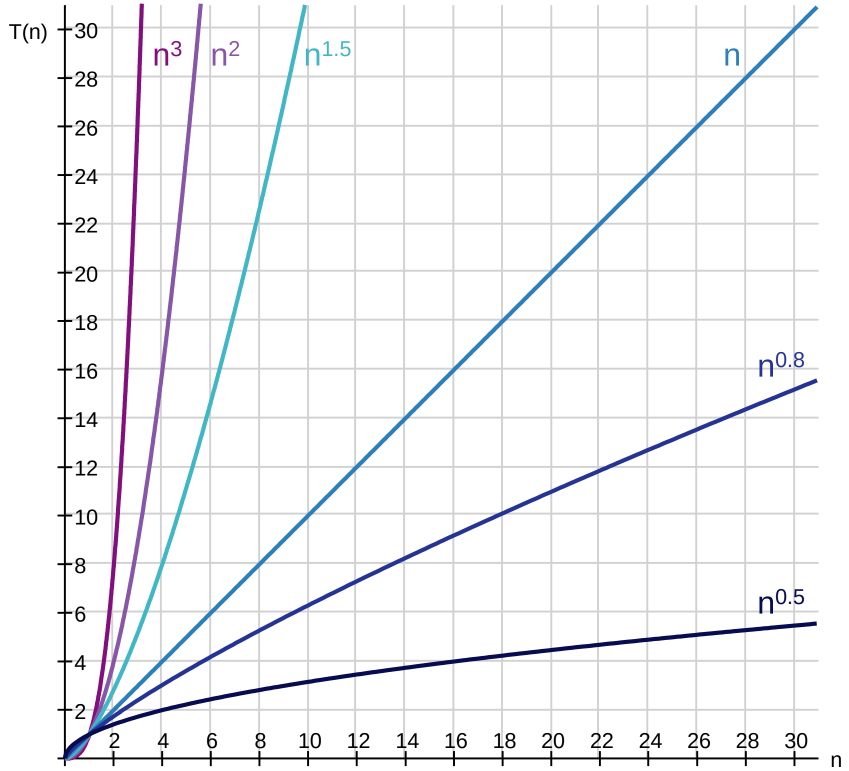
Polynomial family of functions, i.e. functions of type $f(n) = n^k$, $0 < k < \infty$, are all distinct from each other and have growing order of growth with growing exponent («polynomial family of functions»).

$\Theta(n^x) \neq \Theta(n^y)$ iff $x \neq y$

and

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

This also means, that the exponent of the root *does* matter:
$\Theta(\sqrt[2]{n}) \neq \Theta(\sqrt[3]{n})$

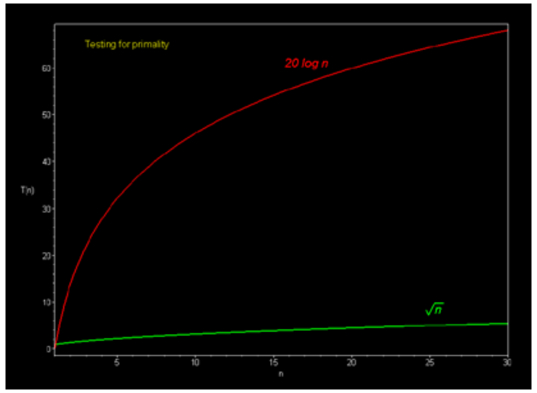


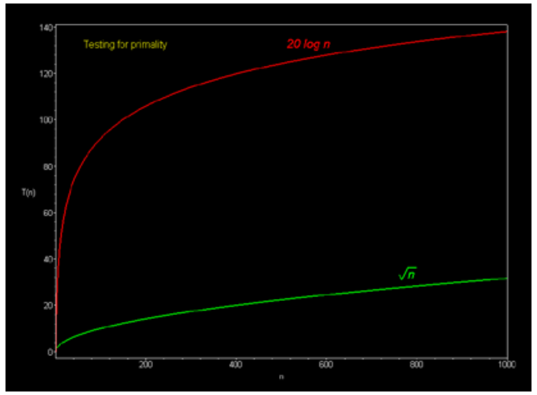
Rules for and some notes on asymptotic complexity

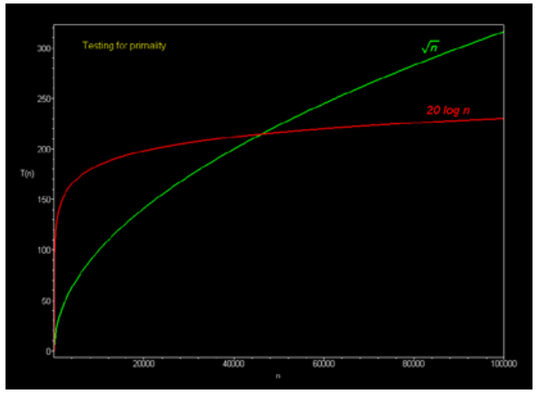
Any polynomial function, i.e. n^k, $0 < k < \infty$, regardless of exponent, grow faster than any logarithmic function.

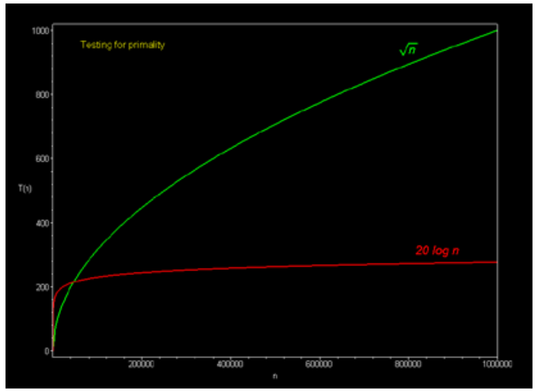
Example: $\sqrt{n}$ grows faster than log(n) asymptotically

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









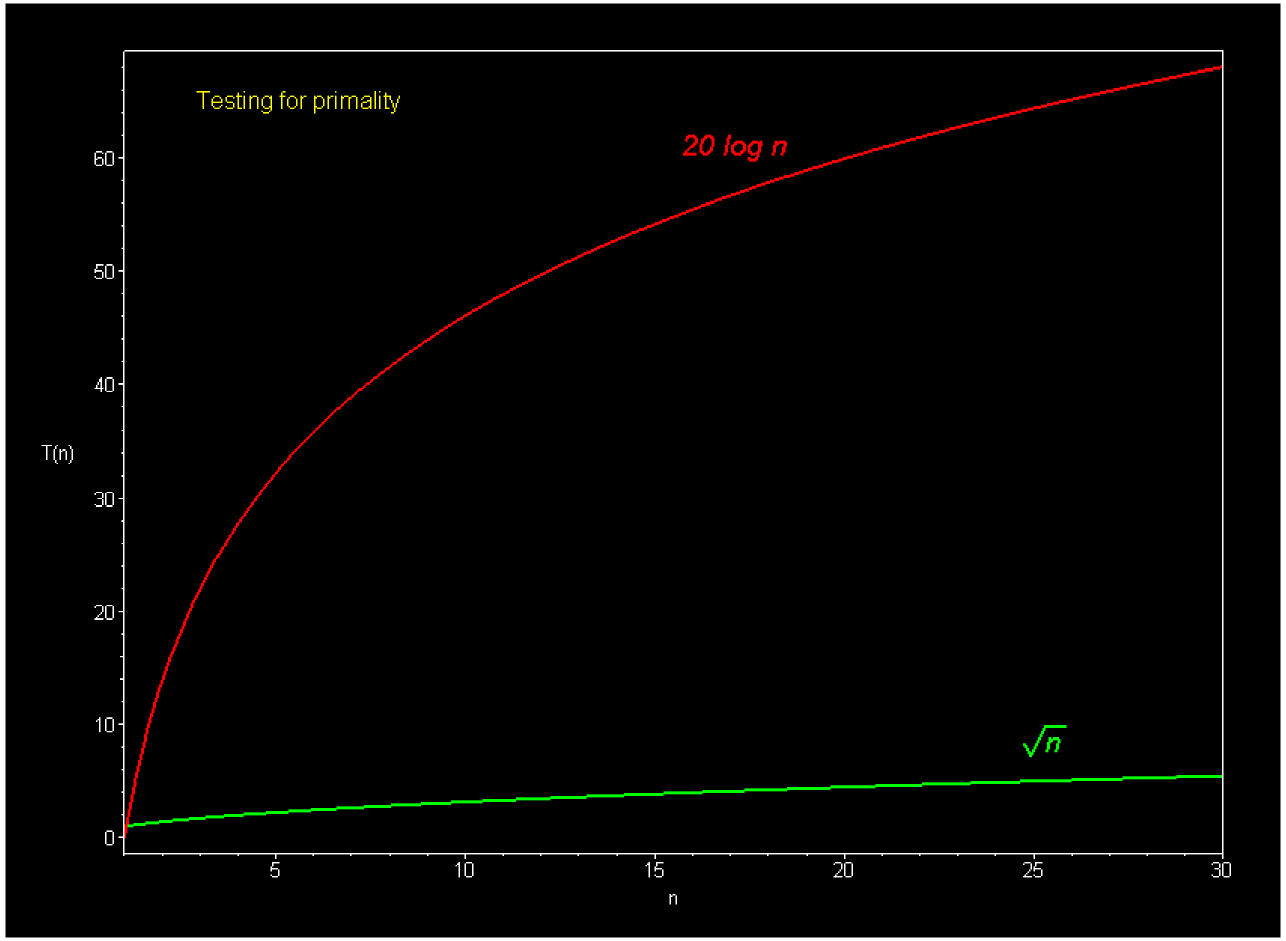
Rules for and some notes on asymptotic complexity

Comparision of

f(n) = 20 log(n)

and

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



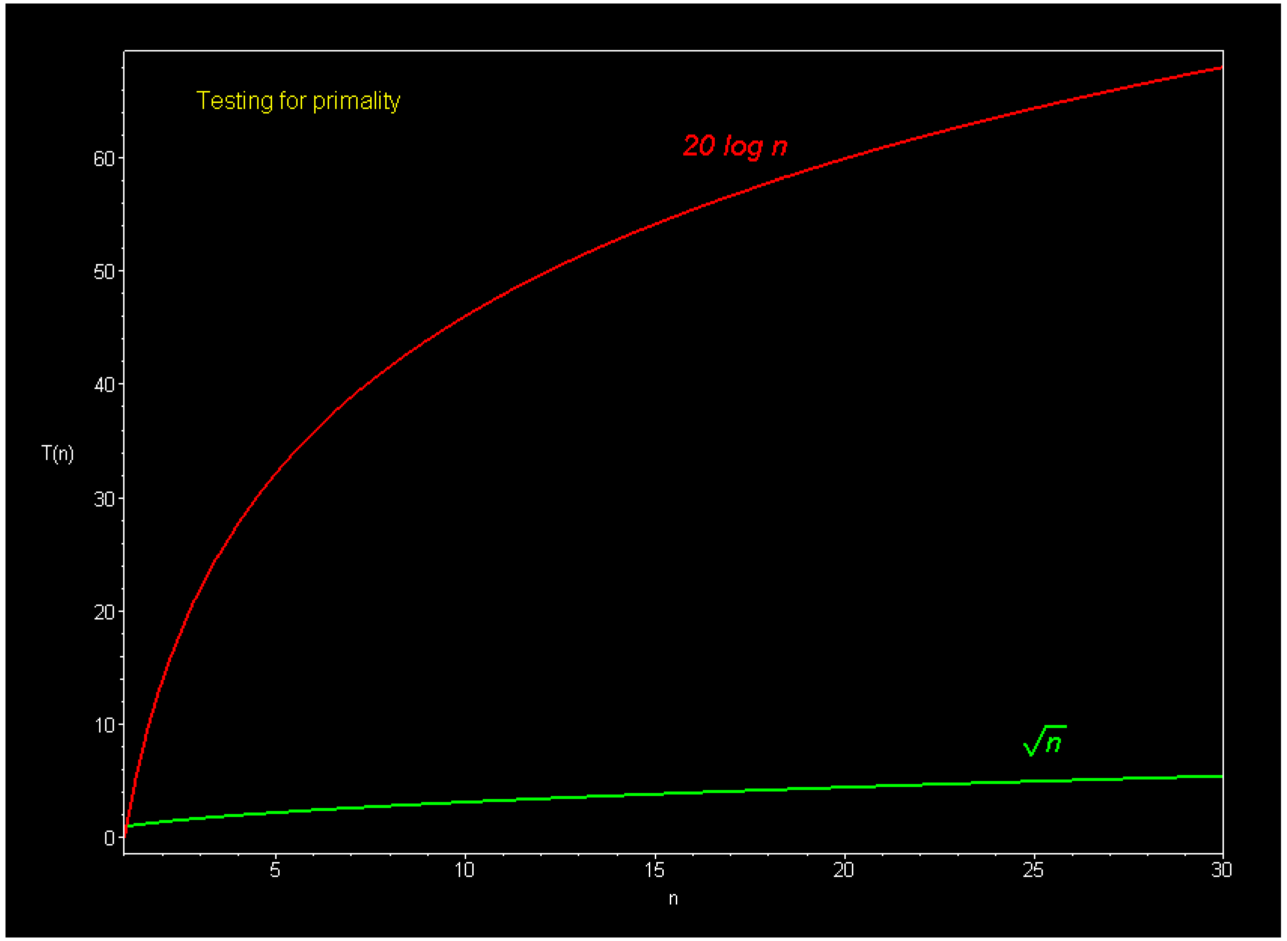
Rules for and some notes on asymptotic complexity

Comparision of

f(n) = 20 log(n)

and

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



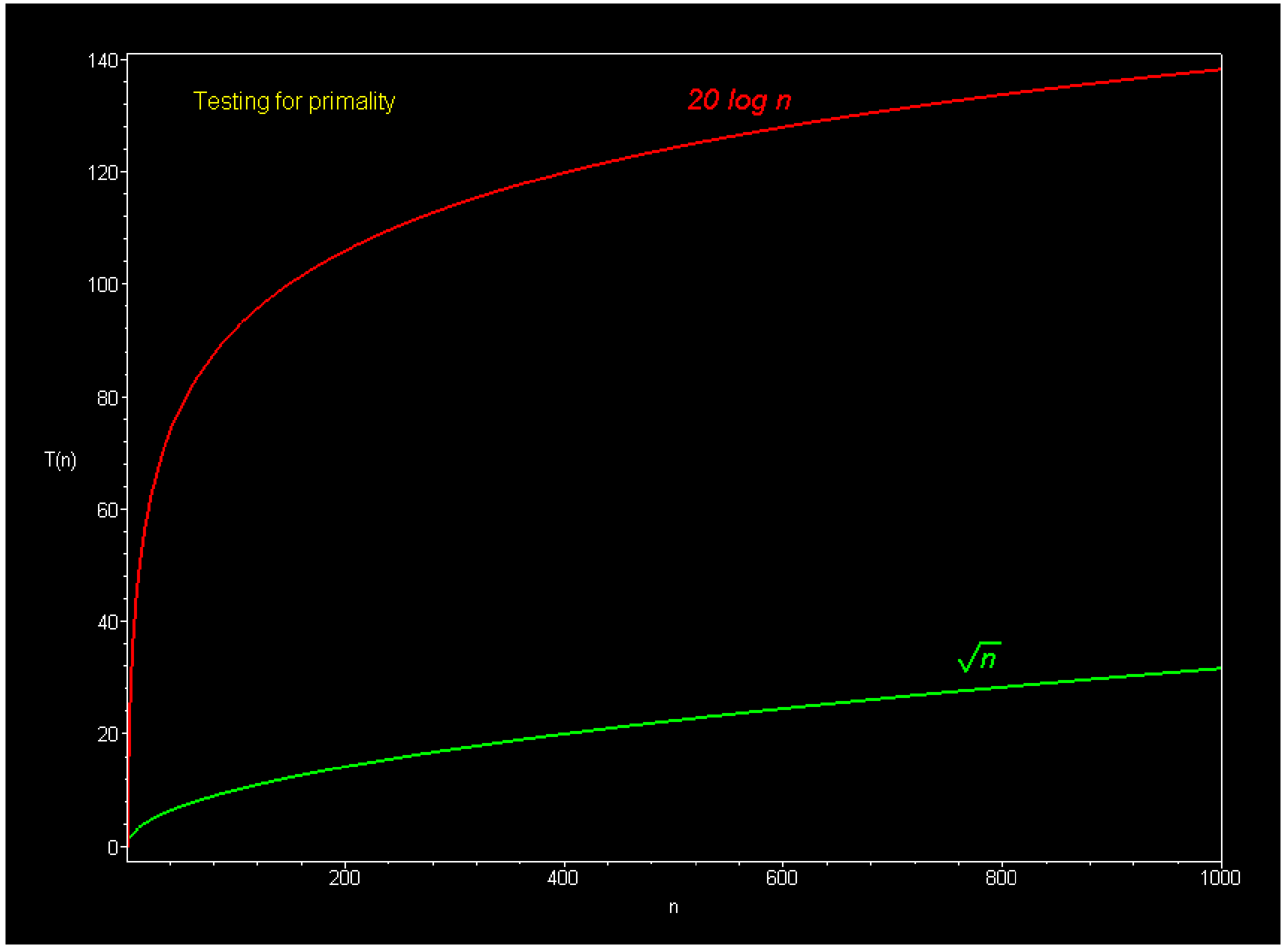
Rules for and some notes on asymptotic complexity

Comparision of

f(n) = 20 log(n)

and

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



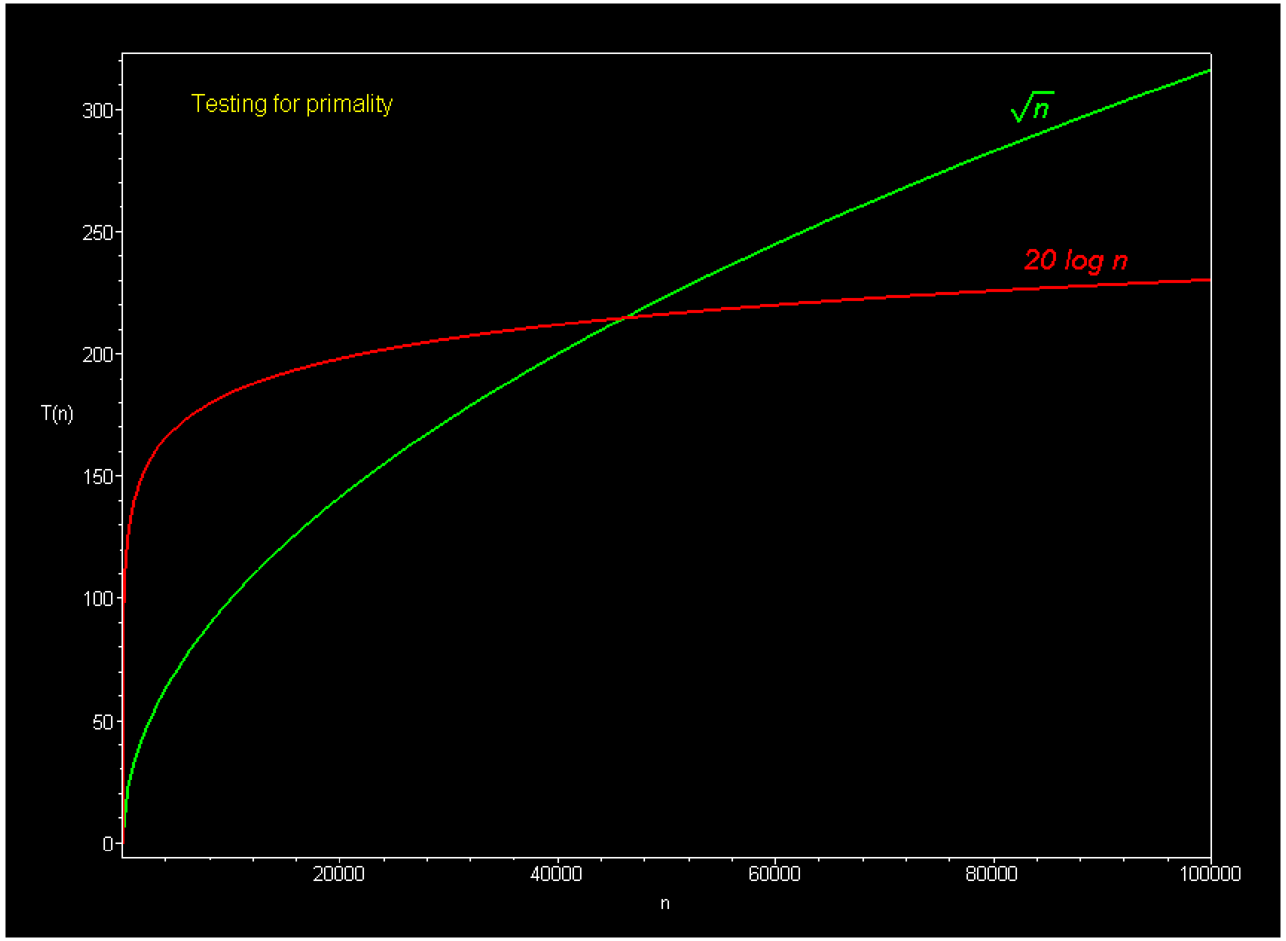
Rules for and Some Notes on Asymptotic Complexity

Comparision 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):

$$\Theta(\log_a(n)) = \Theta(\log_b(n))$$
$$= \Theta(\lg(n)) = \Theta(\ln(n)) = \Theta(\mathrm{ld}(n))$$

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 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:

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

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:

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

– 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 Θ 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…

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

According to l'Hospital's rule, we can write:

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

$$= \limsup_{n \to \infty} \frac{\frac{1}{n}}{\frac{1}{2} \cdot n^{-\frac{1}{2}}} = \limsup_{n \to \infty} \frac{2}{\sqrt{n}} = 0 < \infty$$

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
O	Upper bound	f1 ≤ f2	$0 \leq \limsup_{n \to \infty} \frac{f(n)}{g(n)} = c < \infty$
Ω	Lower bound	f1 ≥ f2	$0 < \liminf_{n \to \infty} \frac{f(n)}{g(n)} \leq \infty$
Θ	Tight bound	f1 = f2	$0 < \lim_{n \to \infty} \frac{f(n)}{g(n)} = c < \infty$
o	Strict upper bound	f1 < f2	$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$
ω	Strict lower bound	f1 > f2	$\limsup_{n \to \infty} \frac{f(n)}{g(n)} = \infty$

Asymptotic complexity: additional examples

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

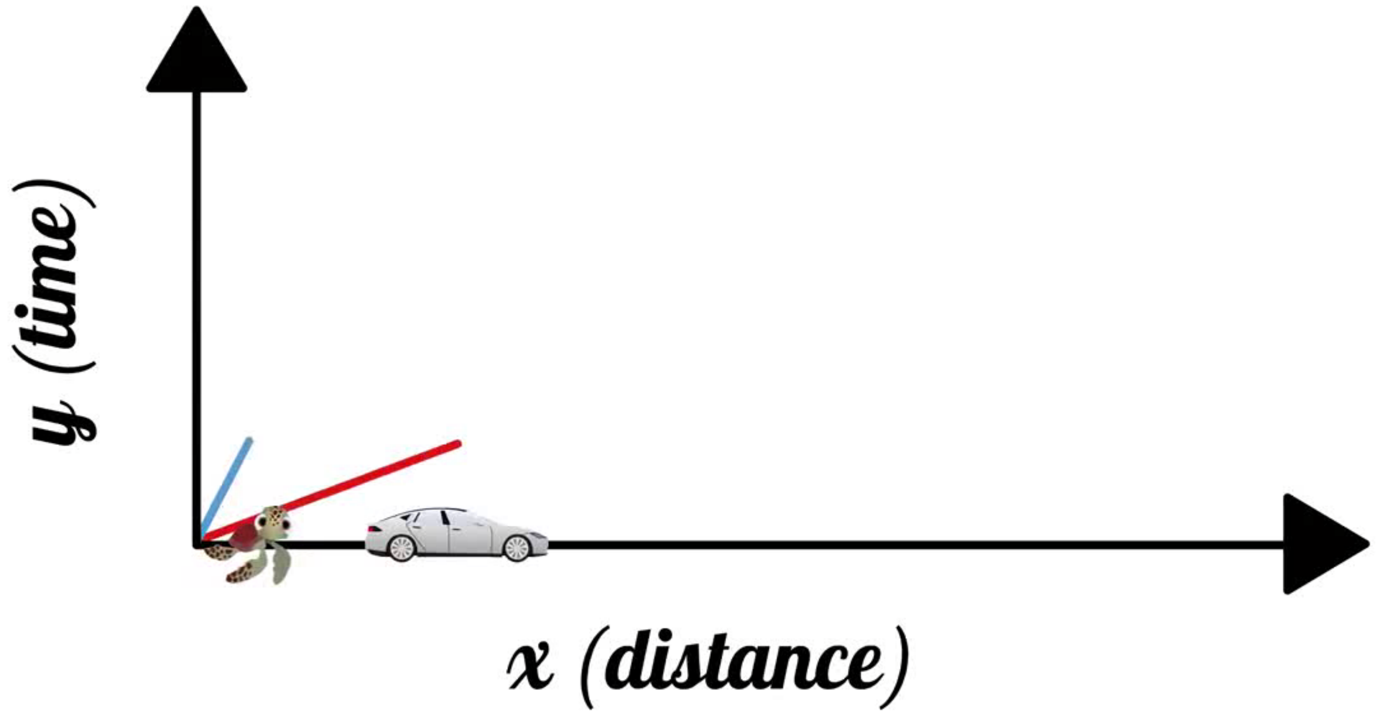
- $f_1(n) = n^{10} \cdot \sqrt[4]{n}$
- $f_2(n) = \log\left(n^{\pi} \cdot \sqrt[2]{n}\right)$
- $f_3(n) = n^{15} \cdot \sqrt[6]{n} \cdot \log(n)$
- $f_4(n) = \log_{\ln(5)}((\log(n))^{\lg(100)})$
- $f_5(n) = (\sqrt{2})^{\log_2(n)} + \log(n)$
- $f_6(n) = \log(n!) + n \cdot \log^5(n)$
- $f_7(n) = (⅔)^n$

Asymptotic Complexity: 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 vmax = 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 importants 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

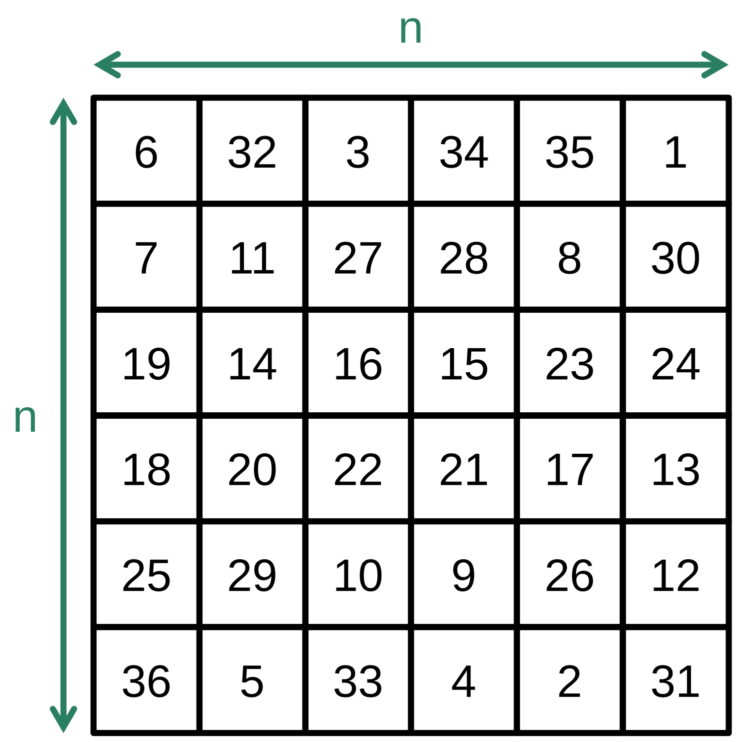
Universität Zürich^UZH

Institut für Informatik

Asymptotic Complexity: Defining the Input

Consider a square A[1..n][1..n] of n times n integers with N = n * n entries.

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



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++)   O(n)
        {
                for (int j = n; j > n - i; j--)   O(n)
                {
                        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)       O(log n)
        {
                for (int j = 1; j <= n; j++)       O(n)
                {
                        printf("Hello World!");
                }
        }
}
```

O(n 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++) {    O(n)
                for (int j = 0; j < n; j++) {    O(n)
                        for (int k = 0; k < j; k++) {    O(n)
                                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++)    O(only once)
        {
                for (int j = n; j > 1; j = j / 2)  O(log n)
                {
                        printf(j);
                }
        }
}
```

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!");
                }
        }
}
```

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;
                }
        }
}
```

O(1)

Running Time Analysis: Additional Exercise

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

Algorithm Running Time Analysis

- General Principles
- Examples Solved

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: findKmax(A, k, n)

```
for i = 1 to n − k + 1 do
    max = A[i]
    for j = 1 to k – 1 do
        if A[i+ j] > max then
            max = A[i + j]
    print(max)
```

Algorithm Running Time Analysis: Example 1 Solution

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

```
1  for i = 1 to n − k + 1 do
2      max = A[i]
3      for j = 1 to k – 1 do
4          if A[i+ j] > max then
5              max = A[i + j]
6      print(max)
```

$$T(n) = c_1 \cdot (n - k + 2) + c_2 \cdot (n - k + 1) + c_3 \cdot k \cdot (n - k + 1) + c_4 \cdot (k - 1) \cdot (n - k + 1) +$$

$$+ c_5 \cdot \alpha \cdot (k - 1) \cdot (n - k + 1) + c_6 \cdot (n - k + 1) =$$

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

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.

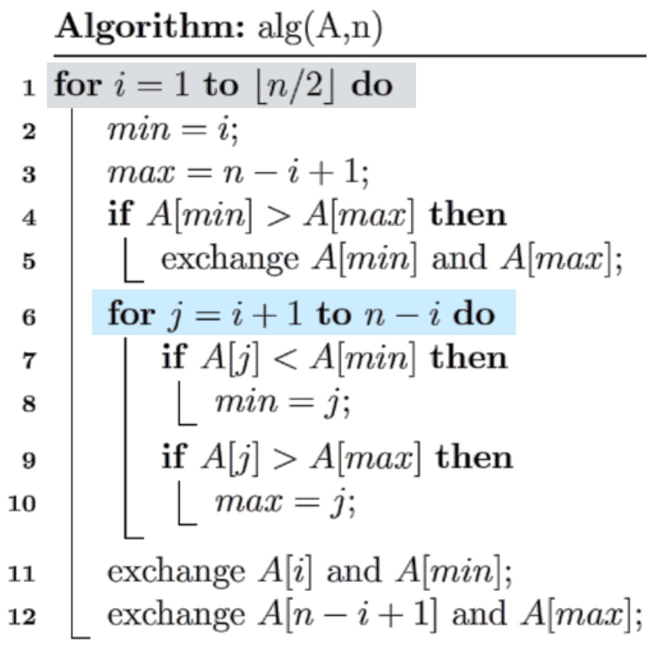
Algorithm: alg(A, n)

```
1   for i = 1 to ⌊n / 2⌋ 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 following example shows how to analyse the number executions of the body of the inner loop (lines 7 to 10) of the algorithm 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…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
⋮	⋮	⋮
k	k+1 … n−k	n − 2·k
⋮	⋮	⋮
⌊n/2⌋	⌊n/2⌋+1 … ⌊n/2⌋	0



```
Algorithm: 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: alg(A,n)

```
for $i = 1$ to $\lfloor n/2 \rfloor$ do
    $min = i$;
    $max = n - i + 1$;
    if $A[min] > A[max]$ then
        exchange $A[min]$ and $A[max]$;
    for $j = i + 1$ to $n - i$ do
        if $A[j] < A[min]$ then
            $min = j$;
        if $A[j] > A[max]$ then
            $max = j$;
    exchange $A[i]$ and $A[min]$;
    exchange $A[n - i + 1]$ and $A[max]$;
```

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):

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

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

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

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

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

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:

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

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

Summations: Polynomial Expressions

– Sum of squares:

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

– Sum of cubes:

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

– Sum of fourths:

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

Summations: Polynomial Expressions

– Sum of the square of a sum:

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

Summations: Geometric Series

A summation of the form

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

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

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

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

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

Summations: Harmonic Series

A summation of the form

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

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

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

(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)`?

```c
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;
}
```

$\rightarrow$ **O(n^3)**

$$\sum_{i=1}^{n} \sum_{j=0}^{n} \sum_{k=1}^{j} 1 =$$

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

$$= n \cdot \frac{n \cdot (n+1)}{2} = ½ \cdot (n^3 + n^2) = f_K(n)$$

$\rightarrow f_K(100) = 505000$

Summations: Decomposing Sums

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

$$\sum_{k=x}^{n-x} k = \underbrace{\sum_{k=1}^{n-x} k}_{\text{sum up «too much», i.e. start summing up from 1 instead of starting from x}} - \underbrace{\sum_{k=1}^{x-1} k}_{\text{remove what has been summed up too much, i.e. numbers from 1 to x − 1}}$$

This is useful in solving sums stemming from nested loops with interacting loop variables like for example:
<pseudo code>

Im Falle, dass über eine Konstante aufsummiert wird, gilt einfach ENDE – ANFANG + 1

Universität Zürich^UZH

Institut für Informatik

Preview on Exercise 3

- Task 1: Running Time Analysis
- Task 2: Asymptotic Tight Bounds
- Task 3: Special Case Analysis

Exercise 3 – Task 1: Running Time Analysis

Algorithm `whatDoesItDo(A,k)` gets an array `A[1..n]` of n integers as an input.

a) Implement the algorithm as a C program that reads the elements of A and prints the result.

b) Describe what the algorithm does.

c) Do an exact analysis of the running time of the algorithm.

d) Determine the best and the worst case of the algorithm. What is the running time and asymptotic complexity in each case?

e) What influence has the parameter k in the asymptotic complexity?

```
Algo: WHATDOESITDO(A,k)

sum = 0;
for $i = 1$ to $k$ do
    mini = i;
    for $j = i + 1$ to $length(A)$ do
        if $A[j] < A[mini]$ then
            mini = j;
    sum = sum + A[mini];
    swp = A[i];
    A[i] = A[mini];
    A[mini] = swp;
return $sum$
```

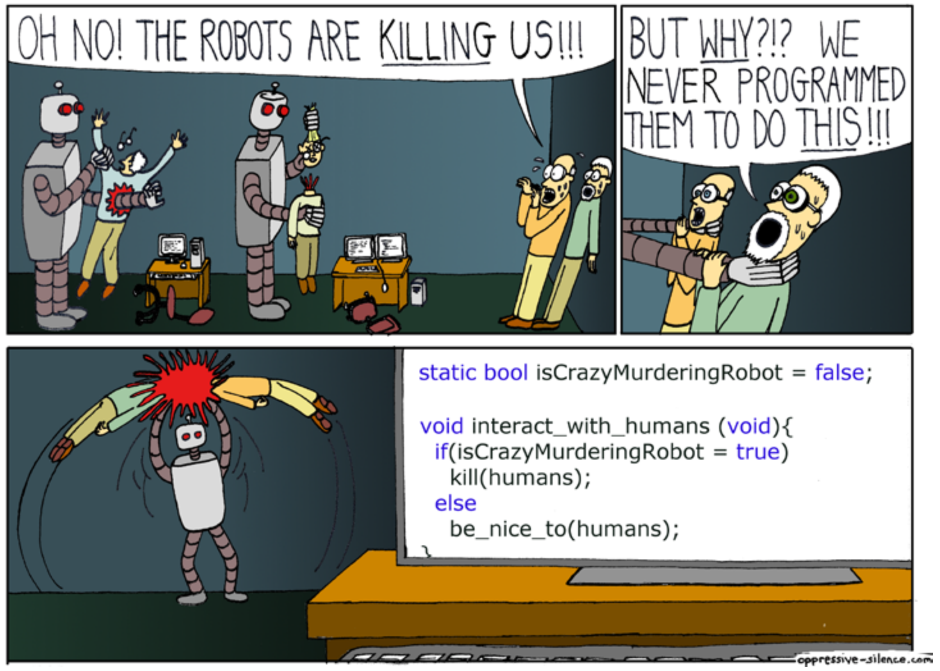
Some Pitfalls When Converting C to Pseudocode

Converting pseudo code into a high level computer programming language is in general straight forward. But there are some pitfalls:

– Pseudocode may start indexing of arrays with 1 whereas C code starts indexing at 0.

– Pseudocode is using = for comparisons while C code is using ==.

  For example, while (I = max) will lead to problems that can be difficult to track. (Use the -Wall compiler flag to get warned if you are compiling on the shell. An IDE may warn you depending on your settings.) And by the way, this is a quite nasty error also without being used in context of pseudocode conversion.


OH NO! THE ROBOTS ARE KILLING US!!!
BUT WHY?!? WE NEVER PROGRAMMED THEM TO DO THIS!!!
oppressive-silence.com

```
static bool isCrazyMurderingRobot = false;

void interact_with_humans (void){
  if(isCrazyMurderingRobot = true)
    kill(humans);
  else
    be_nice_to(humans);
}
```

(from http://blog.acthompson.net/2017/06/student-programmer-fix-this-code.html)

Exercise 3 – Task 2: Asymptotic Tight Bounds

Calculate the asymptotic tight bound for the following functions and rank them by their order of growth (lowest first). Clearly work out the calculation steps in your solution.

$$f_1(n) = n^n + 2^{2n} + 13^{124}$$
$$f_2(n) = \log\left(14(n-1)n^{3n+2}\right)$$
$$f_3(n) = 4^{log_2 n}$$
$$f_4(n) = 12\sqrt{n} + 10^{223} + \log 5^n$$
$$f_5(n) = n^2 \log(n+1) + n \log n^2 + 0.5n$$
$$f_6(n) = 7n^4 + 100n \log n + \sqrt{32} + n$$
$$f_7(n) = \log\left(\min\left(n, \sqrt{n}\right)\right)$$
$$f_8(n) = \log^2(n) + 50\sqrt{n} + \log(n)$$
$$f_9(n) = (n+3)!$$
$$f_{10}(n) = 2\log(6^{\log n^2}) + \log(\pi n^2) + n^3$$

Exercise 3 – Task 2: Asymptotic Tight Bounds

- Attention: In this task, tight bounds are requested. Use Θ notation.
- No formal proofs are necessary in this task. Just apply algebraic transformation into simpler functions for which you know the complexity class and which can be compared easily. Show your work (calculations used for transformation).
- Note that where the base of the logarithms is not given, it (should) not matter. Put differently: If the base of a logarithm matters, it will be given (but there might by tasks where it is given even though it does not matter).
- This is a standard exam question which will be in the first midterm with high probabilty. These are points which you should gather.

Exercise 3 – Task 3: Special Case Analysis

Given two strings A and B, develop an algorithm that checks if B is a substring of A and, if so, returns the number of occurrences of B in A.

a) Specify all the special cases that need to be considered and provide examples of the input data for each of them.

b) Write a C program implementing your algorithm and make sure it runs for all the special cases you provided. Include a function `int substrings(char A[], char B[])` which returns the number of occurrences of B in A. Your program should print the number of occurrences along with the starting and ending index of each occurrence. Here is an example output corresponding to the call `substrings([nitrite], [it])`:

```
A = [nitrite]
B = [it]
Indices = (2,3) (5,6)
Repetitions = 2
```

You are not allowed to use string-functions and/or `string.h`.

Universität Zürich^UZH

Institut für Informatik

Roundup

- Summary
- Outlook
- Questions

Outlook

Next tutorial: Friday, 13.03.2020, Y35-F-32

Topics:

- Review of Exercise 3
- Preview to Exercise 4
- Recurrences
- … (your wishes)

Questions?

Thank you for your attention.