COMPLEXITY

Algorithms and Data Structures 1
Exercise – 2023S

Markus Jäger (Computer Science)
Florian Beck (Artificial Intelligence)
Raja Zafar (Artificial Intelligence)

Institute of Pervasive Computing
Johannes Kepler University Linz

markus.jaeger@jku.at
florian.beck@jku.at
raja.zafar@pervasive.jku.at

JOHANNES KEPLER UNIVERSITÄT LINZ
Altenberger Straße 69
4040 Linz, Österreich
jku.at

COMPLEXITY (ASYMPTOTIC BEHAVIOUR)

- Motivation
- Definition

- Big-O Notation
  - Rules
  - Examples

- Recurrence Systems
  - Unfolding + Examples
  - Master Theorem + Examples

COMPLEXITY :: MOTIVATION

Algorithm analysis is essential for understanding algorithms well enough, in order to apply them to practical problems:

- Performance of a certain algorithm (worst- / best- / average-case)?
- Runtime behaviour?
- Behaviour in a new environment?

Many algorithms are based on the **principle of recursive decomposition**:

- A large problem is broken down into several smaller problems, and
- the solutions of the partial problems are used for solving the original problem
- E.g.: QuickSort, MergeSort, binary search, etc.

COMPLEXITY :: O-NOTATION

For describing the asymptotic time complexity we use the **O-notation** (by Landau)

- Rough measure for the runtime of an algorithm
- Variable factors (such as the problem size n) are considered to be **aiming towards infinity**
- Constant factors, as well as terms whose orders are smaller than the determining term, are neglected, e.g.:

  $O(2n) \rightarrow O(n)$

  $O(n^2 + n) \rightarrow O(n^2)$

COMPLEXITY :: TYPICAL COMPLEXITIES

O(f(n))	Growth	
O(1)	constant	*excellent*
O(log n)	logarithmic	

COMPLEXITY :: TYPICAL COMPLEXITIES

O(f(n))	Growth	
O(1)	constant	*excellent*
O(log n)	logarithmic	
O(n)	linear	*acceptable*
O(n * log n)	slightly over linear	

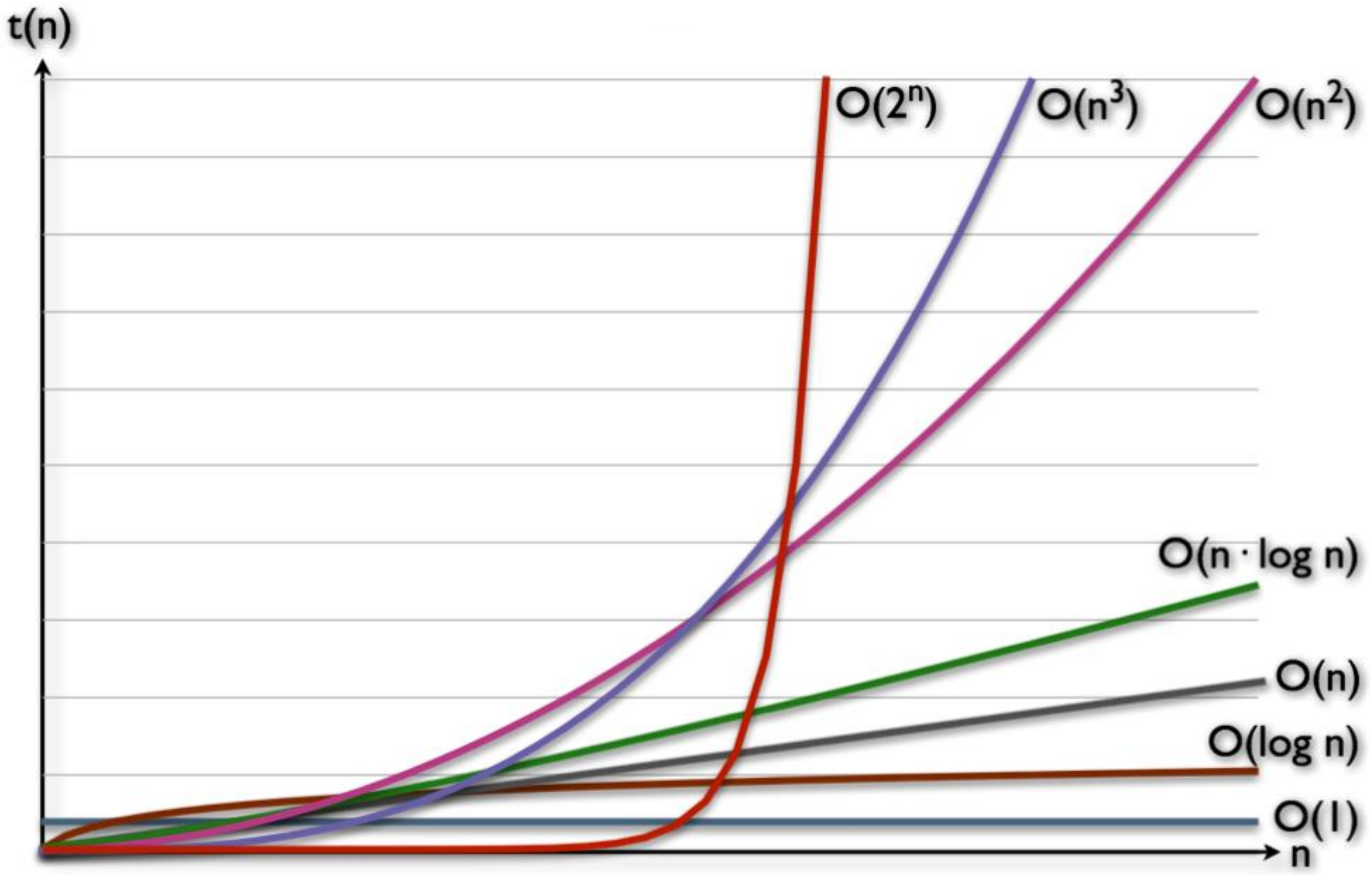
COMPLEXITY :: TYPICAL COMPLEXITIES

O(f(n))	Growth	
O(1)	constant	*excellent*
O(log n)	logarithmic	
O(n)	linear	*acceptable*
O(n * log n)	slightly over linear	
$O(n^2)$	quadratic	*bad*
$O(n^3)$	cubic	
$O(n^k)$	polynomial	

COMPLEXITY :: TYPICAL COMPLEXITIES

O(f(n))	Growth	
O(1)	constant	*excellent*
O(log n)	logarithmic	
O(n)	linear	*acceptable*
O(n * log n)	slightly over linear	
$O(n^2)$	quadratic	*bad*
$O(n^3)$	cubic	
$O(n^k)$	polynomial	
$O(2^n)$	exponential	*disastrous*

COMPLEXITY :: TYPICAL COMPLEXITIES



COMPLEXITY :: … IN COMPUTER SCIENCE

Legend

Excellent | Good | Fair | Bad | Horrible

Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	-	O(1)	O(1)	O(1)	-	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)

Source: http://bigocheatsheet.com/

COMPLEXITY :: … IN COMPUTER SCIENCE

Legend

Excellent | Good | Fair | Bad | Horrible

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	O(n log(n))	O(n log(n))	O(n^2)	O(log(n))
Mergesort	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Timsort	O(n)	O(n log(n))	O(n log(n))	O(n)
Heapsort	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Shell Sort	O(n)	O((nlog(n))^2)	O((nlog(n))^2)	O(1)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)

Graph Operations

Node / Edge Management	Storage	Add Vertex	Add Edge	Remove Vertex	Remove Edge	Query
Adjacency list	$O(\lvert V\rvert+\lvert E\rvert)$	O(1)	O(1)	$O(\lvert V\rvert + \lvert E\rvert)$	$O(\lvert E\rvert)$	$O(\lvert V\rvert)$
Incidence list	$O(\lvert V\rvert+\lvert E\rvert)$	O(1)	O(1)	$O(\lvert E\rvert)$	$O(\lvert E\rvert)$	$O(\lvert E\rvert)$
Adjacency matrix	$O(\lvert V\rvert^2)$	$O(\lvert V\rvert^2)$	O(1)	$O(\lvert V\rvert^2)$	O(1)	O(1)
Incidence matrix	$O(\lvert V\rvert \cdot \lvert E\rvert)$	$O(\lvert V\rvert \cdot \lvert E\rvert)$	$O(\lvert V\rvert \cdot \lvert E\rvert)$	$O(\lvert V\rvert \cdot \lvert E\rvert)$	$O(\lvert V\rvert \cdot \lvert E\rvert)$	$O(\lvert E\rvert)$

Source: http://bigocheatsheet.com/

COMPLEXITY :: RULES

1. **Constant factors are not considered**
   $O(2n) = O(n)$

2. **$T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$**
   $T_1(n) + T_2(n) = Max(O(f(n)), O(g(n)))$
   $T_1(n) * T_2(n) = O(f(n) * g(n))$

3. **T(n) is a polynomial of the order x: $T(n) = (n+1)^x$**
   $T(n) = O(n^x)$

COMPLEXITY :: CALCULATION

A low-level analysis resulted in the following expression:

$$T(n) = 4n\ (-2 + 3n)\ (n - ld(n))\ /\ n$$

What is the asymptotic time complexity?

$$
\begin{aligned}
T(n) &= (-8n + 12n^2)\ (n - ld(n))\ /\ n \\
&= (-8n^2 + 12\ n^3 + 8n\ ld(n) - 12n^2\ ld(n))\ /\ n \\
&= -8n + 12n^2 + 8\ ld(n) - 12n\ ld(n) \\
&\mathbf{= 12n^2 + 8\ ld(n) - 8n - 12n\ ld(n)}
\end{aligned}
$$

→ **asymptotic runtime complexity: $O(n^2)$**

COMPLEXITY :: EXAMPLES

for-loops

```
for i in range(n):
  for j in range(n):
    k = k + 1
```

$O(n^2)$

```
for i in range(n):
  for j in range(5):
    # do something
```

$O(5n) = O(n)$

COMPLEXITY :: EXAMPLES

branches (if-then-else)

```
if (condition):   # e.g. O(1)
    Statement1    # e.g. O(n)
else:
    Statement2    # e.g. O(n²)
```

$O(\max(O(1), O(n), O(n^2))) = O(n^2)$

Sequence of statements

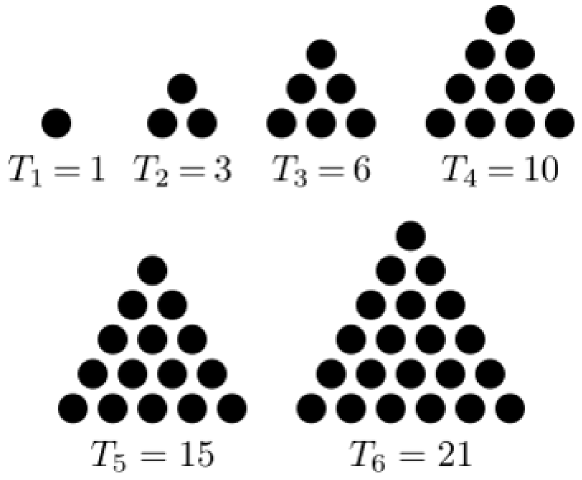
```
for i in range(n):
  x[i] = 0
for i in range(n):
  for j in range(n):
    x[i] = x[i] + 1
```

$O(n)+O(n^2) = O(n^2)$

COMPLEXITY :: EXAMPLES

Nested for-loops

```
for i in range(n):
    for j in range(i):
        # Statements
```



	j=1	j=2	j=3	j=4	∑
i=1	*	-	-	-	1
i=2	*	*	-	-	3
i=3	*	*	*	-	6
i=4	*	*	*	*	10

→ Asymptotic time complexity?

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2)$$

COMPLEXITY :: LOGARITHMIC COMPLEXITY

Logarithmic complexities can be found where the problem size is continuously divided by a certain factor:

- divide & conquer - algorithms
- **The divisor is the basis of the logarithm**

```
12345678        n = 8
1234            n1 = n/2 = 4
12              n2 = n/(2*2) = n/4 = 2
1               n3 = n/(2*2*2) = n/8 = n/n = 1
```

- Reason: How often do you have to divide until you reach a single element?

$n/2^p = 1 \rightarrow n = 2^p \rightarrow p = \log_2(n) \quad \log_2(n) = ld(n)$

Hints: $\log_k(n) = \log(n)/\log(k)$

$\log_b(a) = x \Rightarrow a = b^x$

COMPLEXITY :: EXAMPLE

Find the complexities!

Example 1 $O(n * \log(n))$

```
i = 1
while i <= n:
  j = n
  while j > 1:
    j = j / 10
  i = i + 1
```

Example 2 $O(n^2 * \log_2(n))$

```
for i in range(n):
  for j in range(n):
    k = n
    while k > 1:
      k = k / 2
```

RECURRENCE SYSTEMS

RECURRENCE SYSTEMS :: IN GENERAL

Recurrence systems describe the **runtime behaviour of recursive algorithms** (e.g., MergeSort):

- **Divide**: Reduce problem (the sequence to be sorted is split into partial sequences)
- **Conquer**: sort the parts (e.g. recursive use of MergeSort)
- **Combine**: Reading subsequence simultaneously and mixing them, by reading the smallest element of each subsequence and writing it to a new sequence

Pseudocode:

```
algorithm MergeSort(S) → S_S
      Input: sequence S
      Output: sorted sequence S_S

   if S is only one element
     return S
   else
     divide S in 2 halves S_1 and S_2;  //DIVIDE
     S_1  := MergeSort(S_1);            //CONQUER
     S_2  := MergeSort(S_2);            //CONQUER
   return Merge(S_1, S_2);              //COMBINE
```

Runtime behaviour of MergeSort:

$$T(n) = 2 * T(n/2) + n, \; n \geq 2$$

$$T(1) = 1$$

RECURRENCE SYSTEMS :: O-NOTATION

Asymptotic runtime complexity

Only the time behaviour of the algorithm for a potentially infinitely large input quantity is of interest

f = O(g)... „f does not grow faster than g“ (upper limit)

$$\exists\ c \in \mathit{R}^{+},\ n_0 \in N_0\ ,\ \forall\ n \geq n_0\colon f(n) \leq c\ g(n)$$

f = Ω(g)… „f grows at least as fast as g” (lower limit)

$$g = O(f)$$

f = Θ(g)… „f and g are of the same growth order“ (exactly)

$$f = O(g) \text{ und } g = O(f)$$

RECURRENCE SYSTEMS :: ANALYSIS (UNFOLDING)

Principle: Repeated insertion until a regularity is discovered

1) Insertion

```
T(n) = 2T(n/2) + n, n ≥ 2
T(1) = 1
---------------------------------------
T(n) = 2T(n/2)          + n  =
     = 2(2T(n/4)+n/2) + n  = 4T(n/4)   + 2n =
     = 4(2T(n/8)+n/4) + 2n = 8T(n/8)   + 3n =
     = 8(2T(n/16)+n/8)+ 3n = 16T(n/16)+ 4n =
     = ... =
```

$\underline{T(n) = 2^i * T(n/2^i) + i*n}$

2) Maximum recursion depth

At which i $(n/2^i = 1)$ is the maximum recursion depth reached?

→ i = ld(n), as for $n/2^{ld(n)} = n/n = 1$

RECURRENCE SYSTEMS :: ANALYSIS (UNFOLDING)

3) Insert i (i = ld(n))

```
T(n) = 2^ld(n) * T(n/2^ld(n)) + ld(n) * n =
     = n * T(n/n)             + ld(n) * n =
     = n * T(1)               + ld(n) * n =
     = n + ld(n) * n
```

4) Asymptotic runtime complexity

f = O(g) ... „f does not grow faster than g“ (upper asymptotic limit), if $\exists\, c \in R^+, n_o \in N_0, \forall n \geq n_0: f(n) \leq c\, g(n)$

Assumption: T(n) = O(n * ld(n))
Proof: It is sufficient to find any c for which an n_0 exists, from which **f(n) = n + n*ld(n) ≤ c*n*ld(n)** with increasing n

e.g.: c = 2:

```
n + n * ld(n) ≤ 2 * n * ld(n) →
            n ≤ n * ld(n)     →
            1 ≤ ld(n)         →
            2 ≤ n
T(n) = O(n + n * ld(n)) = O(n * ld(n))
```

RECURRENCE SYSTEMS :: ANALYSIS (GUESS & PROOF)

Principle: Guess the solution and proof it using induction

1) Guess

```
T(n) = 2T(n/2) + n, n ≥ 2
T(1) = 1
-----------------------------------------
→ T(n) = O(n + n * ld(n)) → O(n * ld(n))
```

2) Proof using induction

Induction assumption: T(n) = n + n * ld(n)
Induction base: T(1) = 1 + 1 * ld(1) = 1 → OK
Induction step: insert induction assumption

```
T(n) = 2T(n/2) + n
     = 2(n/2 + (n/2) * ld(n/2)) + n
     = n + n * ld(n/2) + n
     = 2n + n * (ld(n) - ld(2))
     = 2n + n * ld(n) - n
     = n + n * ld(n)
```

RECURRENCE SYSTEMS :: EXAMPLES (UNFOLDING + PROOFING)

T(1) = 7
T(n) = T(n/5) + 7 for n ≥ 2

Solution (unfolding):

$T(n) = T(n/5) + 7,\ n \geq 2$
$T(1) = 7$

$T(n) = T(n/5) + 7 =$
$= (T((n/5)/5) + 7) + 7 =$
$= T(n/25) + 14 =$
$= (T((n/25)/5) + 7) + 14 =$
$= T(n/125) + 21 = \ldots =$
$T(n) = T(n/5^i) + 7i$

max. recursion depth at **$i = \log_5 n$**

$T(n) = T(n/5^i) + 7i$
$= T(n/5^{\log_5 n}) + 7*\log_5 n$
$= T(n/n) + 7*\log_5 n$
$= T(1) + 7*\log_5 n$
$= 7 + 7*\log_5 n$

$\underline{T(n) = O(\log n)}$

Solution (proofing):

$T(n) = T(n/5) + 7,\ n \geq 2$
$T(1) = 7$

Induction assumption: $T(n) = 7+7*\log_5 n$
Induction base: $T(1) = 7+7*\log_5 1 = 7+7*0$ **OK**
Induction step: $T(n) = T(n/5) + 7 =$
$= (7+7*\log_5(N/5))+7 =$

(we know: $\log_5(n/5) = \log_5(n) - \log_5 5 = \log_5(n) - 1$)

$= 7+7(\log_5 n-1)+7 =$
$= 7+7*\log_5 n-7 + 7 =$

$\underline{= 7+7*\log_5 n}$

RECURRENCE SYSTEMS :: ANALYSIS (MASTER THEOREM)

Principle: Approach for recurrence equations of the form: `T(n) = a * T(n/b) + f(n)`
with $a \geq 1$, $b > 1$ and $f(n)$ asymptotic positive

- If $\exists\ \varepsilon > 0$, such that $f(n) = O(n^{(\log_b a)-\varepsilon})$, then
  $T(n) = \Theta(n^{\log_b a})$
- If $\exists\ k \geq 0$, such that $f(n) = \Theta(n^{\log_b a} \log^k n)$, then
  $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
- If $\exists\ \varepsilon > 0$ and $c < 1$, such that $f(n) = \Omega(n^{(\log_b a)+\varepsilon})$ and $a\, f(n/b) \leq c\, f(n)$ for n sufficiently large, then
  $T(n) = \Theta(f(n))$

$f = O(g)$	f does not grow faster than g
$f = \Theta(g)$	f and g are of the same order
$f = \Omega(g)$	f grows at least as fast as g

RECURRENCE SYSTEMS :: EXAMPLE (MASTER THEOREM)

`T(n) = 2T(n/2) + n, n ≥ 2`

Solution:

$a = 2$
$b = 2$
$f(n) = n$
$n^{\log_b a} = n^{\log_2 2} = n^1 = n$

$f(n)$ compare with $n^{\log_b a}$
$f(n) = \Theta(n^{\log_b a}) = \Theta(n)$
→ For k=0, n and n are of the same order → **case 2**

$T(n) = \Theta(n^{\log_b a} \log n) =$ **$\underline{\Theta(n \log n)}$**

`T(n) = a T(n/b) + f(n)`

case 1 $\exists\ \varepsilon > 0: f(n) = O(n^{(\log_b a) - \varepsilon})$
$T(n) = \Theta(n^{\log_b a})$

case 2 $\exists\ k \geq 0: f(n) = \Theta(n^{(\log_b a)} \log^k n)$
$T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

case 3 $\exists\ \varepsilon > 0, c < 1: f(n) = \Omega(n^{(\log_b a) + \varepsilon})$ and $a\,f(n/b) \leq c\,f(n)$ for n sufficiently large, then
$T(n) = \Theta(f(n))$

$f = O(g)$ **f does not grow faster than g**
$f = \Theta(g)$ **f and g are of the same order**
$f = \Omega(g)$ **f grows at least as fast as g**

RECURRENCE SYSTEMS :: EXAMPLE (MASTER THEOREM)

`T(n) = 8T(n/2) + 1/n⁴`

Solution:

$a = 8$
$b = 2$
$f(n) = 1/n^4 = n^{-4}$
$n^{\log_b a} = n^{\log_2 8} = n^3$

$f(n)$ compare with $n^{\log_b a}$ results in **case 1**:

$f(n) = O(n^{(\log_b a)-\varepsilon}) = \Theta(n^{3-\varepsilon})$
→ n^{-4} does not grow faster than $n^{3-\varepsilon}$
→ Is true for a constant $\varepsilon > 0$ (e.g.: $\varepsilon=7$)

$T(n) = \Theta(n^{\log_b a}) =$ **$\Theta(n^3)$**

`T(n) = a T(n/b) + f(n)`

case 1 $\exists\ \varepsilon > 0$: $f(n) = O(n^{(\log_b a)-\varepsilon})$
$T(n) = \Theta(n^{\log_b a})$

case 2 $\exists\ k \geq 0$: $f(n) = \Theta(n^{(\log_b a)} \log^k n)$
$T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

case 3 $\exists\ \varepsilon > 0, c < 1$: $f(n) = \Omega(n^{(\log_b a)+\varepsilon})$ and $a\ f(n/b) \leq c\ f(n)$ for n sufficiently large, then
$T(n) = \Theta(f(n))$

$f = O(g)$	**f does not grow faster than g**
$f = \Theta(g)$	**f and g are of the same order**
$f = \Omega(g)$	**f grows at least as fast as g**

RECURRENCE SYSTEMS :: EXAMPLE (MASTER THEOREM)

`T(n) = 16T(n/4) + n^2.5`

Solution:

$a = 16$
$b = 4$
$f(n) = n^{2.5}$
$n^{\log_b a} = n^{\log_4 16} = n^2$

$f(n)$ compare with $n^{\log_b a}$ results in **case 3**:

$f(n) = \Omega(n^{(\log_b a)+\varepsilon}) = \Omega(n^{2+\varepsilon})$
→ $n^{2.5}$ grows at least as fast as $n^{2+\varepsilon}$
→ is true for a constant $\varepsilon > 0$ (e.g. $\varepsilon=0.5$)

Proof of the additional condition:
$a * f(n/b) = 16(n/4)^{2.5} = 16n^{2.5}/4^{2.5} =$
$n^{2.5}(1/4^{0.5}) = f(n) * (1/4^{0.5})$ → $\underline{c = (1/4^{0.5}) < 1}$

$T(N) = \Theta(f(n)) =$ $\underline{\boldsymbol{\Theta(n^{2.5})}}$

`T(n) = a T(n/b) + f(n)`

case 1 $\exists\, \varepsilon > 0$: $f(n) = O(n^{(\log_b a)-\varepsilon})$
$T(n) = \Theta(n^{\log_b a})$

case 2 $\exists\, k \geq 0$: $f(n) = \Theta(n^{(\log_b a)} \log^k n)$
$T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

case 3 $\exists\, \varepsilon > 0, c < 1$: $f(n) = \Omega(n^{(\log_b a)+\varepsilon})$ and $a\, f(n/b) \leq c\, f(n)$ for n sufficiently large, then
$T(n) = \Theta(f(n))$

$f = O(g)$	**f does not grow faster than g**
$f = \Theta(g)$	**f and g are of the same order**
$f = \Omega(g)$	**f grows at least as fast as g**

COMPLEXITY

Algorithms and Data Structures 1
Exercise – 2023S

Markus Jäger (Computer Science)
Florian Beck (Artificial Intelligence)
Raja Zafar (Artificial Intelligence)

Institute of Pervasive Computing
Johannes Kepler University Linz

markus.jaeger@jku.at
florian.beck@jku.at
raja.zafar@pervasive.jku.at

JOHANNES KEPLER UNIVERSITÄT LINZ
Altenberger Straße 69
4040 Linz, Österreich
jku.at