Complexity

Laura Kovács

Example of a Program Code

How many elementary operations are performed at most when executing the code for input of a given size n?

```
public static long factorial (int n)
{
    long factorial := 1;
    int i:=1;
    while ( i≤n ) do
      factorial := factorial * i;
      i := i+1;
    end while
    return factorial;
}
```

Example of a Program Code

How many elementary operations are performed at most when executing the code for input of a given size n?

```
1   public static long factorial (int n)
2   {
3           long factorial := 1;
4           int i:=1;
5           while ( i≤n ) do
6               factorial := factorial * i;
7               i := i+1;
8           end while
9           return factorial;
11  }
```

Line	Number of elementary operations
3	1
4	1
5	4
6	4
7	3
9	1

For example, in **line 6** we:

-look up the value of *i*;

-look up the value of *factorial;*

-multiply these two values;

-assign the result of multiplication to *factorial.*

Example of a Program Code

How many elementary operations are performed at most when executing the code for input of a given size n?

```
1   public static long factorial (int n)
2   {
3           long factorial := 1;
4           int i:=1;
5           while ( i≤n ) do
6               factorial := factorial * i;
7               i := i+1;
8           end while
9           return factorial;
11  }
```

Line	Number of elementary operations	How often is executed
3	1	1
4	1	1
5	4	n
6	4	n
7	3	n
9	1	1

Example of a Program Code

How many elementary operations are performed at most when executing the code for input of a given size n?

```
1   public static long factorial (int n)
2   {
3           long factorial := 1;
4           int i:=1;
5           while ( i≤n ) do
6               factorial := factorial * i;
7               i := i+1;
8           end while
9           return factorial;
11  }
```

Line	Number of elementary operations	How often is executed
3	1	1
4	1	1
5	4	n
6	4	n
7	3	n
9	1	1

Total number of elementary operations:

$$\mathbf{f: \mathbb{N} \rightarrow \mathbb{N}, f(n) =} \ 1+1+4*n+4*n+3*n+1 = \mathbf{11*n+3}$$

Example of a Program Code

How many elementary operations are performed at most when executing the code for input of a given size n?

```
1   public static long factorial (int n)
2   {
3           long factorial := 1;
4           int i:=1;
5           while ( i≤n ) do
6               factorial := factorial * i;
7               i := i+1;
8           end while
9           return factorial;
11  }
```

Line	Number of elementary operations	How often is executed
3	1	1
4	1	1
5	4	n
6	4	n
7	3	n
9	1	1

An estimate of the

Total number of elementary operations:

$f: \mathbb{N} \rightarrow \mathbb{N}$, $f(n)$ = $1+1+4*n+4*n+3*n+1$ = **$11*n +3$**

Number of elementary operations performed by a computer depend on many factors!

For example:

- operation on a LONG/INT type can get 2 elementary operations, instead of 1 ⇒ **11*n+6** is also reasonable.

Example of a Program Code

How many elementary operations are performed at most when executing the code for input of a given size n?

```
1   public static long factorial (int n)
2   {
3           long factorial := 1;
4           int i:=1;
5           while ( i≤n ) do
6               factorial := factorial * i;
7               i := i+1;
8           end while
9           return factorial;
11  }
```

Line	Number of elementary operations	How often is executed
3	1	1
4	1	1
5	4	n
6	4	n
7	3	n
9	1	1

An estimate of the

Total number of elementary operations:

$$\mathbf{f: \mathbb{N} \rightarrow \mathbb{N}, f(n) =} 1+1+4*n+4*n+3*n+1 = \mathbf{11*n +3}$$

UPPER BOUND abstraction for all estimates f: Big O Notation

Big O Notation - Definition

Let ℕ be the set of natural numbers.

Let f: ℕ → ℕ and g: ℕ → ℕ be two functions.

Then **$f \in O(g)$** iff:

$$\exists\, c, n_0: \ (c, n_0 \in \mathbb{N}) \wedge (c>0) : \Big(\forall n: (n \in \mathbb{N} \wedge n \geq n_0): \ (f(n) \leqslant c * g(n) \Big)$$

We say:

- O(g) is the order of function g (Ordnung of g)
- If $f \in O(g)$, then f is of the Order of g (von der Ordnung g)

Big O Notation - Definition

Let $\mathbb{N}$ be the set of natural numbers.

Let f: $\mathbb{N} \rightarrow \mathbb{N}$ and g: $\mathbb{N} \rightarrow \mathbb{N}$ be two functions.

Then **$f \in O(g)$** iff:

$$\exists\ c, n_0:\ (c, n_0 \in \mathbb{N}) \wedge (c>0) : \Big(\ \forall n: (n \in \mathbb{N} \wedge n \geq n_0):\ (f(n) \leqslant c^* g(n))\Big)$$

We say:

- O(g) is the order of function g (Ordnung of g)
- If $f \in O(g)$, then f is of the Order of g (von der Ordnung g)

Note: If g: $\mathbb{N} \rightarrow \mathbb{N}$ with g(n)=n, we write $f \in O(n)$ instead of $f \in O(g)$.

Note: If $g(n) \neq 0$ for every $n \in \mathbb{N}$, then $f \in O(g) \Leftrightarrow \lim_{n \rightarrow \infty} f(n)/g(n) = c$

Big O Notation - Definition

Let ℕ be the set of natural numbers.

Let f: ℕ → ℕ and g: ℕ → ℕ be two functions.

Then **f∈O(g)** iff:

$$\exists\, c, n_0\!: \ (c, n_0 \in \mathbb{N}) \wedge (c>0) : \Big(\forall n\!: (n \in \mathbb{N} \wedge n \geq n_0)\!: \ (f(n) \leqslant c{*}g(n) \Big)$$

Example **(Revisited Factorial Example from slides 2-6):**

Consider f: ℕ → ℕ, f(n)=11*n+3. Then f∈?

Big O Notation - Definition

Let $\mathbb{N}$ be the set of natural numbers.

Let f: $\mathbb{N} \rightarrow \mathbb{N}$ and g: $\mathbb{N} \rightarrow \mathbb{N}$ be two functions.

Then **$f \in O(g)$** iff:

$$\exists\, c, n_0\colon\ (c, n_0 \in \mathbb{N}) \wedge (c>0) : \Big(\forall n\colon (n \in \mathbb{N} \wedge n \geq n_0)\colon\ (f(n) \leqslant c * g(n)\Big)$$

Example **(Revisited Factorial Example from slides 2-6)**:

Consider f: $\mathbb{N} \rightarrow \mathbb{N}$, $f(n)=11*n+3$. Then $f \in O(n)$.

Proof: We choose $c=12$, $n_0=3$. It remains to show:

$\forall n\colon (n \in \mathbb{N} \wedge n \geq 3)\colon\ (11*n+3 \leqslant 12*n)$, that is $11*n+3 \leqslant 12*n$ for all $n \geq 3$.

Since $n \geq 3$, we have $11*n + \underline{3} \leqslant 11*n+n = 12*n$.

Therefore, $11*n+3 \in O(n)$.

Big O Notation - Definition

Let $\mathbb{N}$ be the set of natural numbers.

Let f: $\mathbb{N} \rightarrow \mathbb{N}$ and g: $\mathbb{N} \rightarrow \mathbb{N}$ be two functions.

Then **$f \in O(g)$** iff:

$$\exists\, c, n_0:\ (c, n_0 \in \mathbb{N}) \wedge (c>0) : \Big(\forall n: (n \in \mathbb{N} \wedge n \geq n_0):\ (f(n) \leqslant c * g(n)\Big)$$

Example (Revisited Factorial Example from slides 2-6):

- Consider f: $\mathbb{N} \rightarrow \mathbb{N}$, $f(n)=11*n+3$. Then $f \in O(n)$.
- Consider f_1 : $\mathbb{N} \rightarrow \mathbb{N}$, $f_1(n)=11*n+6$. Then $f_1 \in O(n)$.
- Consider f_2: $\mathbb{N} \rightarrow \mathbb{N}$, $f_2(n)=7*n+1$. Then $f_2 \in O(n)$.

In case of the Factorial Example:

<u>At most O(n)</u> elementary operations are performed when executing the code <u>for input n</u>.

Big O Notation - Properties

Computer Programs implement Algorithms.

Algorithms can be:

- deterministic - non-deterministic
  (deterministisch) (nichtdeterministisch)
- sequential - parallel
  (sequentiell) (parallel)
- finite - infinite
  (endlich) (undendlich)
- reversible - irreversible
  (reversibel) (irreversibel)

Big O Notation - Properties

Computer Programs implement Algorithms.

Algorithms can be *implemented and executed in different ways*, depending on computer properties.

(16 / 32/ 64 bits memory allocation for input, CPU-cycles, memory/time limit, etc.)

O(·) measures the worst-case (ungünstigsten Fall) *complexity* *(Aufwand)* of an ALGORITHM!

O(·) gives an *upper bound* on the execution time of an ALGORITHM!

O(·) does not depend on computer properties!

O(·) depends only on the INPUT of the ALGORITHM!

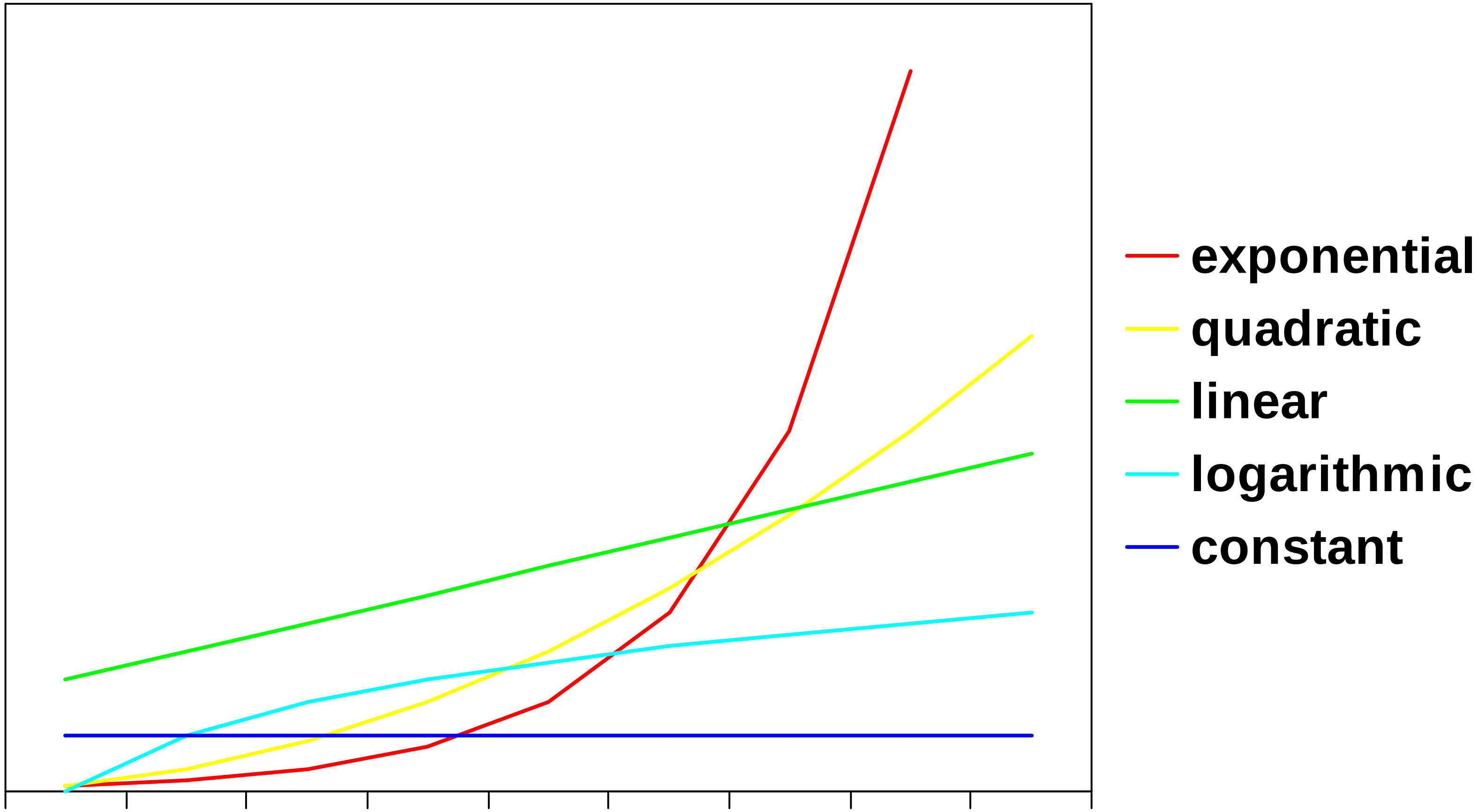
Algorithms can be:

- deterministic - non-deterministic
  (deterministisch) (nichtdeterministisch)
- sequential - parallel
  (sequentiell) (parallel)
- finite - infinite
  (endlich) (undendlich)
- reversible - irreversible
  (reversibel) (irreversibel)

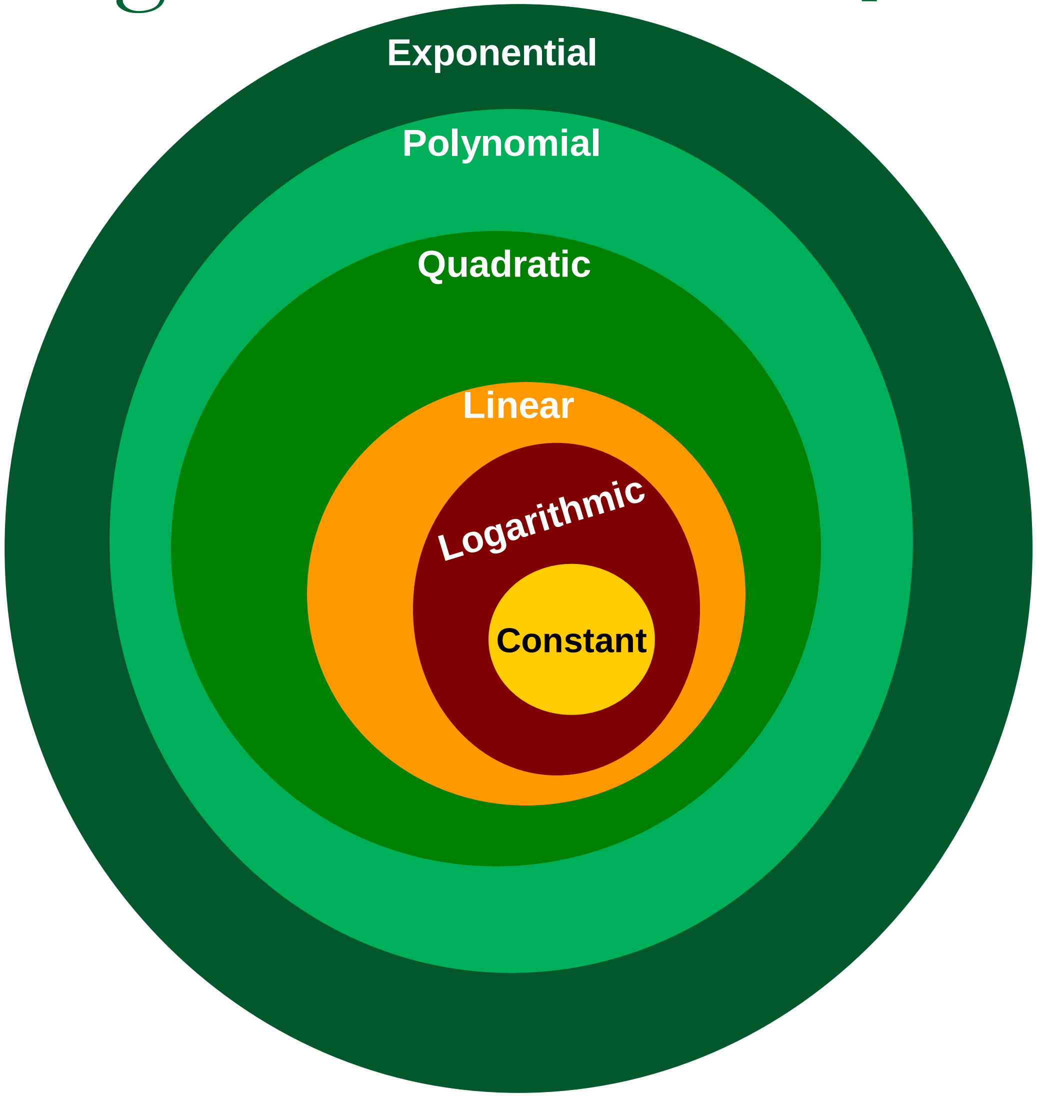
Big O Notation – Some Common Orders

$O(1)$	Constant (konstant) If $g: \mathbb{N} \to \mathbb{N}$ with $g(n)=1$, we write $f \in O(1)$ instead of $f \in O(g)$.
$O(\log n)$	Logarithmic (logarithmisch) If $g: \mathbb{N} \to \mathbb{N}$ with $g(n)=\log n$, we write $f \in O(\log n)$ instead of $f \in O(g)$.
$O(n)$	Linear (linear) If $g: \mathbb{N} \to \mathbb{N}$ with $g(n)=n$, we write $f \in O(n)$ instead of $f \in O(g)$.
$O(n^2)$	Quadratic (quadratisch) If $g: \mathbb{N} \to \mathbb{N}$ with $g(n)=n^2$, we write $f \in O(n^2)$ instead of $f \in O(g)$.
$O(n^k)$ $k \in \mathbb{N}$	Polynomial (polynomial) If $g: \mathbb{N} \to \mathbb{N}$ with $g(n)=n^k$, we write $f \in O(n^k)$ instead of $f \in O(g)$.
$O(e^n)$	Exponential (exponentiell) If $g: \mathbb{N} \to \mathbb{N}$ with $g(n)=e^n$, we write $f \in O(e^n)$ instead of $f \in O(g)$.

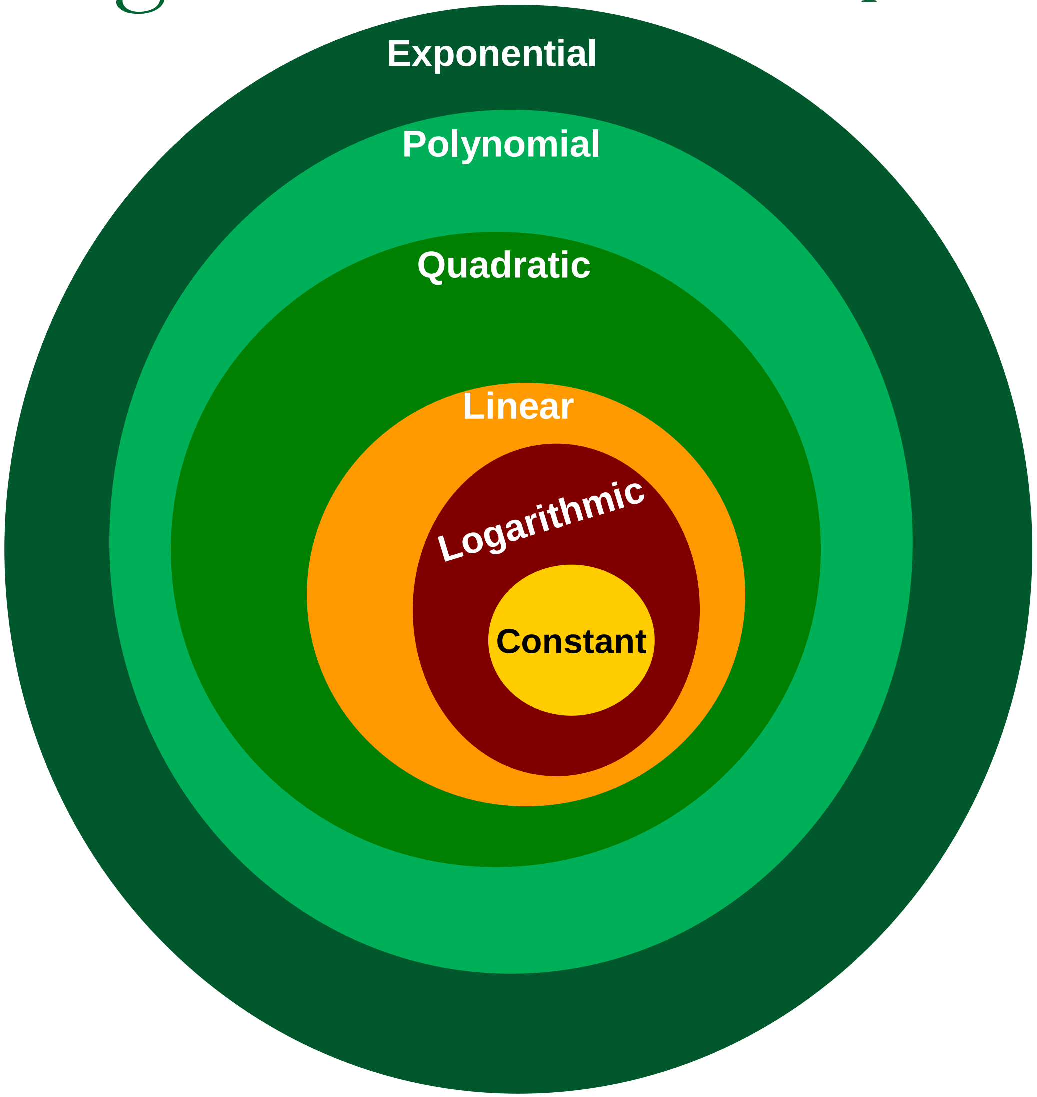
Big O Notation – Some Common Orders



Big O Notation – Dependency among Common Orders



Big O Notation – Dependency among Common Orders



Note:

If f∈O(1), then clearly:

- f∈O(log n);
- f∈O(n).
- f∈O(n^2);
- f∈O(n^k);
- f∈O(e^n).

BUT, we are interested in giving a

TIGHTEST

complexity approximation

Examples of Complexity Measurements for Algorithms with input n

n	ld n	n ld n	n^2	2^n	n!
10	**3**	**33**	**100**	**1024**	**$3*10^6$**
20	**4**	**86**	**400**	**10^6**	**$2*10^{18}$**
100	**7**	**664**	**10’000**	**10^{31}**	**10^{161}**
1000	**10**	**10.000**	**10^6**		
10000	**13**	**130.000**	**10^8**		

Big O Notation – Calculus Rules

Let $f: \mathbb{N} \to \mathbb{N}$ and $g: \mathbb{N} \to \mathbb{N}$ be two functions.

- ❑ $O(c*f) = O(f)$, where **$c \in \mathbb{N}$** is a **constant**
- ❑ $O(f) + O(g) = O(f+g)$

  Note: $O(f+g) = \max\{O(f), O(g)\}$
- ❑ $O(f*g) = O(f)*O(g)$

Some further properties (follows from O-definition):

- **$O(f) \subseteq O(g) \iff f \in O(g)$**
- **$O(f) = O(g) \iff (O(f) \subseteq O(g)) \wedge (O(g) \subseteq O(f))$**
- **$O(f) \subset O(g) \iff (O(f) \subseteq O(g)) \wedge (O(g) \neq O(f))$**

Example:

- $O(\log n) \subset O(n)$
- $O(n*\log n) \subset O(n^2)$
- $O(\log n) \subset O(n1/2$

Big O Notation – Calculus Rules

Let f: $\mathbb{N} \to \mathbb{N}$ and g: $\mathbb{N} \to \mathbb{N}$ be two functions.

- $O(c*f) = O(f)$, where **$c \in \mathbb{N}$** is a **constant**
- $O(f) + O(g) = O(f+g)$

Note: $O(f+g) = \max\{O(f), O(g)\}$

- $O(f*g) = O(f)*O(g)$

Some further properties (follows from O-definition):

Abbreviation: Denoting $\subseteq$ by $\leqslant$

- **$O(f) \subseteq O(g) \;\Leftrightarrow\; f \in O(g)$**
  $O(f) \leqslant O(g) \;\Leftrightarrow\; f \in O(g)$
- **$O(f) = O(g) \Leftrightarrow (O(f) \subseteq O(g)) \wedge (O(g) \subseteq O(f))$**
  $O(f) \leqslant O(g) \;\Leftrightarrow\; f \in O(g)$
- **$O(f) \subset O(g) \Leftrightarrow (O(f) \subseteq O(g)) \wedge (O(g) \neq O(f))$**
  $O(f) < O(g) \;\Leftrightarrow\; (O(f) \subseteq O(g)) \wedge (O(f) \neq O(g))$

Example:

- $O(\log n) \subset O(n)$
- $O(n*\log n) \subset O(n^2)$
- $O(\log n) \subset O(n^{1/2})$

Big O Notation – **Examples**

Estimate the below complexities with O-notation.
The estimation should be as ***tight*** as possible.

- O(2*n-1) = …
- O(n*(n+1)/2) = …
- O(ld n) = …
- O(log n^2) = …
- O((3*n^2 + 6*n+9)*log(1+2*n)) = …

Big O Notation – **Examples**

Estimate the below complexities with O-notation.
The estimation should be as ***tight*** as possible.

- $O(2*n-1) = O(n)$
- $O(n*(n+1)/2) = O(n^2)$
- $O(\mathrm{ld}\ n) = O(\log n)$
- $O(\log n^2) = O(\log n)$
- $O((3*n^2 + 6*n+9)*\log(1+2*n)) = O(n^2 \log(n))$

Lower Bounds of an Algorithm's Execution Time

O-notation for UPPER BOUND (oberen Schrank) **ESTIMATION:**

$\mathbf{f \in O(g)}$ iff $\exists\, c, n_0 \colon\ (c, n_0 \in \mathbb{N}) \wedge (c>0) : (\forall n \colon (n \in \mathbb{N} \wedge n \geq n_0) \colon\ (f(n) \leqslant c * g(n))$

Lower Bounds of an Algorithm's Execution Time

Let $f: \mathbb{N} \to \mathbb{N}$ and $g: \mathbb{N} \to \mathbb{N}$ be two functions.

Then **$f \in \Omega(g)$** iff:

$$\exists\, c, n_0: \ (c, n_0 \in \mathbb{N}) \wedge (c>0) : \Big(\forall n: (n \in \mathbb{N} \wedge n \geq n_0): \ (g(n) \leqslant c*f(n)\Big)$$

O-notation for UPPER BOUND (oberen Schrank, ungünstigsten Fall) **ESTIMATION:**

$f \in O(g)$ iff $\exists\, c, n_0: \ (c, n_0 \in \mathbb{N}) \wedge (c>0) : (\forall n: (n \in \mathbb{N} \wedge n \geq n_0): \ (f(n) \leqslant c*g(n))$

Ω-notation for LOWER BOUND (best-case, unteren Schrank, günstigsten Fall) **ESTIMATION:**

$f \in \Omega(g)$ iff $\exists\, c, n_0: \ (c, n_0 \in \mathbb{N}) \wedge (c>0) : (\forall n: (n \in \mathbb{N} \wedge n \geq n_0): \ (g(n) \leqslant c*f(n))$

Average Bounds of an Algorithm's Execution Time

Example:

$(3n^2+6n+9n)*\log(1+2n) \leqslant 36\, n^2 * \log n \longrightarrow (3n^2+6n+9n)*\log(1+2n) \in O(n^2 * \log n)$

$(3n^2+6n+9n)*\log(1+2n) \geq n^2 * \log n \longrightarrow (3n^2+6n+9n)*\log(1+2n) \in \Omega(n^2 * \log n)$

O-notation for UPPER BOUND (oberen Schrank, ungünstigsten Fall) **ESTIMATION:**

$f \in O(g)$ iff $\exists\, c, n_0$: $(c, n_0 \in \mathbb{N}) \wedge (c>0) : (\forall n: (n \in \mathbb{N} \wedge n \geq n_0)$: $(f(n) \leqslant c*g(n))$

Ω-notation for LOWER BOUND (unteren Schrank, günstigsten Fall) **ESTIMATION:**

$f \in \Omega(g)$ iff $\exists\, c, n_0$: $(c, n_0 \in \mathbb{N}) \wedge (c>0) : (\forall n: (n \in \mathbb{N} \wedge n \geq n_0)$: $(g(n) \leqslant c*f(n))$

Average Bounds of an Algorithm's Execution Time

Example:

$(3n^2+6n+9n)*\log(1+2n) \leq 36\, n^2 * \log n \longrightarrow (3n^2+6n+9n)*\log(1+2n) \in O(n^2 * \log n)$

$(3n^2+6n+9n)*\log(1+2n) \geq n^2 * \log n \longrightarrow (3n^2+6n+9n)*\log(1+2n) \in \Omega(n^2 * \log n)$

$\Downarrow$

$\mathbf{(3n^2+6n+9n)*\log(1+2n) \in \theta\, (n^2 * \log n)}$

O-notation for UPPER BOUND (oberen Schrank, ungünstigsten Fall) ESTIMATION:

$f \in O(g) \quad \text{iff} \quad \exists\, c, n_0\!: (c, n_0 \in \mathbb{N}) \wedge (c>0) : (\forall n\!: (n \in \mathbb{N} \wedge n \geq n_0)\!: (f(n) \leq c*g(n))$

Ω-notation for LOWER BOUND (unteren Schrank, günstigsten Fall) ESTIMATION:

$f \in \Omega(g) \quad \text{iff} \quad \exists\, c, n_0\!: (c, n_0 \in \mathbb{N}) \wedge (c>0) : (\forall n\!: (n \in \mathbb{N} \wedge n \geq n_0)\!: (g(n) \leq c*f(n))$

Average Complexity (mittlere Aufwand): $\mathbf{\theta(g) = O(g) \cap \Omega(g)}$

$f \in \theta(g) \quad \text{iff} \quad \exists\, c_1, c_2, n_0\!: (c_1, c_2, n_0 \in \mathbb{N}) \wedge (c_1>0) \wedge (c_2>0)\!: (\forall n\!: (n \in \mathbb{N} \wedge n \geq n_0)\!: (c_1*g(n) \leq f(n) \leq c_2*g(n))$

Average Bounds of an Algorithm's Execution Time

Factorial Example:

$O(n)$ … *linear* worst-case complexity

$\Omega(1)$ … *constant* best-case complexity

$\theta(n)$ … *linear* average complexity

O-notation for UPPER BOUND (oberen Schrank, ungünstigsten Fall) **ESTIMATION:**

$$f \in O(g) \quad \text{iff} \quad \exists\, c, n_0: \ (c, n_0 \in \mathbb{N}) \wedge (c>0) : (\, \forall n: (n \in \mathbb{N} \wedge n \geq n_0): \ (f(n) \leqslant c * g(n))$$

Ω-notation for LOWER BOUND (unteren Schrank, günstigsten Fall **) ESTIMATION:**

$$f \in O(g) \quad \text{iff} \quad \exists\, c, n_0: \ (c, n_0 \in \mathbb{N}) \wedge (c>0) : (\, \forall n: (n \in \mathbb{N} \wedge n \geq n_0): \ (g(n) \leqslant c * f(n))$$

Average Complexity (mittlere Aufwand): **$\theta(g) = O(g) \cap \Omega(g)$**

$$f \in \theta(g) \quad \text{iff} \quad \exists\, c_1, c_2, n0: \ (c_1, c_2, n0 \in \mathbb{N}) \wedge (c_1>0) \wedge (c_2>0): (\forall n: (n \in \mathbb{N} \wedge n \geq n0): (c_1 * g(n) \leqslant f(n) \leqslant c_2 * g(n))$$

Complexity – Example 1

- Example:

```
boolean f ( int[][] a , int n ) {
        for ( int i = 0 ; i < n ; i++ ) {
                for ( int j = i + 1 ; j < n ; j++ ) {
                        if ( a[i][j] == 0 ) {return false;}
                }
        }
        return true;
}
```

***What is the* worst-case complexity?**

***What is the* best-case complexity?**

***What is the the* average complexity?**

Complexity – Example 1

- Example:

```
boolean f ( int[][] a , int n ) {
        for ( int i = 0 ; i < n ; i++ ) {
                for ( int j = i + 1 ; j < n ; j++ ) {
                        if ( a[i][j] == 0 ) {return false;}
                }
        }
        return true;
}
```

O(n^2) … *quadratic* worst-case complexity
Ω(1) … *constant* best-case complexity
θ(n^2) … *quadratic* average complexity

Complexity – Example 1

- Example:

```
boolean f ( int[][] a , int n ) {
        for ( int i = 0 ; i < n ; i++ ) {
                for ( int j = i + 1 ; j < n ; j++ ) {
                        if ( a[i][j] == 0 ) {return false;}
                }
        }
        return true;
}
```

What is the worst-case complexity in case of all array elements are 1?

Complexity – Example 1

- Example:

```
boolean f ( int[][] a , int n ) {
        for ( int i = 0 ; i < n ; i++ ) {
                for ( int j = i + 1 ; j < n ; j++ ) {
                        if ( a[i][j] == 0 ) {return false;}
                }
        }
        return true;
}
```

What is the worst-case complexity in case of all array elements are 1?

$O(n^2)$

Complexity – Example 2

- Example:

```
boolean f ( int[][] a , int n ) {
        long factorial := 1;
        int i:=1;
        while ( i≤n ) do
            factorial := factorial * i;
            i := i+1;
        end while;

        for ( int i = 0 ; i < n ; i++ ) {
                for ( int j = i + 1 ; j < n ; j++ ) {
                        if ( a[i][j] == 0 ) {return false;}
                }
        }
        return true;
}
```

What is the worst-case complexity?
What is the best-case complexity?
What is the average complexity?

Complexity – Example 2

- Example:

```
boolean f ( int[][] a , int n ) {
        long factorial := 1;
        int i:=1;
        while ( i≤n ) do
            factorial := factorial * i;
            i := i+1;
        end while;

        for ( int i = 0 ; i < n ; i++ ) {
                        for ( int j = i + 1 ; j < n ; j++ ) {
                                        if ( a[i][j] == 0 ) {return false;}
                        }
        }
        return true;
}
```

$O(n^2)$ … *quadratic* worst-case complexity

$\Omega(1)$ … *constant* best-case complexity

$\theta(n^2)$ … *quadratic* average complexity

Big O Notation – Example of Binary Search

$A(n) = 1 + A(n/2)$
$A(1) = 1$

$A(n) = 1 + \mathrm{ld}\ n \;\Rightarrow\; O(A) = O(\log n)$

$A \in O(\log n)$

Big O Notation - P and NP Algorithms

An algorithm (problem) is in **P** iff it can be solved in polynomial time.

An algorithm is in P iff it is solved in O(n^k) steps of execution,
where n is the size of the algorithm's input.

Essentially, **P** corresponds to the class of problems that are realistically **solvable on a computer.**

An algorithm in P is called a P-problem, P-algorithm, polynomial-time problem.

Example: Factorial is in P.

Big O Notation - P and NP Algorithms

An algorithm (problem) is in **P** iff it can be solved in polynomial time.

An algorithm is in P iff it is solved in $O(n^k)$ steps of execution,
where n is the size of the algorithm's input.

Essentially, **P** corresponds to the class of problems that are realistically **solvable on a computer.**

An algorithm in P is called a P-problem, P-algorithm, polynomial-time problem.

An algorithm is in NP iff it can be verified in $O(n^k)$ steps of execution,
where n is the size of the algorithm's input.

For a problem in NP,
- one guesses a solution-candidate;
- verifies (checks) in POLYNOMIAL TIME whether the solution-candidate is indeed a solution.

An algorithm in NP is called an NP-problem, NP-algorithm, nondeterministic polynomial-time problem.

Big O Notation - P and NP Algorithms

An algorithm (problem) is in **P** iff it can be solved in polynomial time.

An algorithm is in P iff it is solved in $O(n^k)$ steps of execution,
where n is the size of the algorithm's input.

Essentially, **P** corresponds to the class of problems that are realistically **solvable on a computer.**

An algorithm in P is called a P-problem, P-algorithm, polynomial-time problem.

$P \subset NP$

An algorithm (problem) is in **NP** iff it can be solved in nondetermistic-polynomial time.

An algorithm is in NP iff it can be verified in $O(n^k)$ steps of execution,
where n is the size of the algorithm's input.

For a problem in NP,
- one guesses a solution-candidate;
- verifies (checks) in POLYNOMIAL TIME whether the solution-candidate is indeed a solution.

An algorithm in NP is called an NP-problem, NP-algorithm, nondeterministic polynomial-time problem.

Open Questions: $P = NP$? $P \neq NP$?

Big O Notation – NP-Complete Problems

Intuitively, A problem is in **NP-complete** iff
-it is in NP
-one cannot do better than NP when solving it.

From one NP-complete problem another NP-complete problem can be obtained in polynomial time.

If one NP-complete problem could be solved in polynomial time, then P=NP.

Big O Notation – Examples of NP-Complete Problems

- **Satisfiability**
- Clique
- Hamiltonian Path
- Graph Coloring
- Subset-sum
- Travelling Salesman
- Scheduling

Satisfiability Problem:

Given a propositional formula with n boolean variables.

Question: Is the formula satisfiable?

Answer: Yes, if the formula is satisfiable;

No, otherwise.

Big O Notation – Examples of NP-Complete Problems

- Satisfiability
- **Clique**
- Hamiltonian Path
- Graph Coloring
- Subset-sum
- Travelling Salesman
- Scheduling

Clique Problem:

Given a graph G and k∈ ℕ

Question: Does G have a k-Clique?

Answer: Yes, if the G has a k-Clique;

No, otherwise.

Big O Notation – Examples of NP-Complete Problems

- Satisfiability
- Clique
- **Hamiltonian Path**
- Graph Coloring
- Subset-sum
- Travelling Salesman
- Scheduling

Hamiltonian Path Problem:

Given a graph G and two nodes u, v of the graph G

Question: Does G have a Hamiltonian Path from u to v?

Answer: Yes, if G has a Hamiltonian Path from u to v;

No, otherwise.

Big O Notation – Examples of NP-Complete Problems

- Satisfiability
- Clique
- Hamiltonian Path
- **Graph Coloring**
- Subset-sum
- Travelling Salesman
- Scheduling

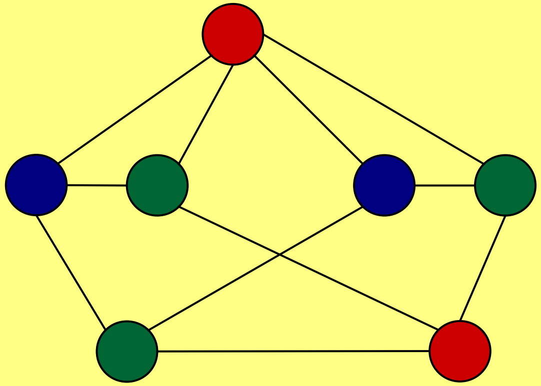
Graph Coloring Problem:

Given a graph G and three distinct colors (Red-Green-Blue)

Question: Can the nodes of G be 3-colored, that is
no two adjacent nodes have the same color?

Answer: Yes, if G can be 3-colored;
No, otherwise.

Example of a 3-colored Graph:



Big O Notation – Examples of NP-Complete Problems

- Satisfiability
- Clique
- Hamiltonian Path
- Graph Coloring
- **Subset-sum**
- Travelling Salesman
- Scheduling

Subset Sum Problem:

Given a set $S=\{x_1,\ldots,x_n\}$ of natural numbers

and a natural number $t \in \mathbb{N}$

Question: Does S have a subset $\{y_1,\ldots,y_k\}$ such that $\Sigma y_i=t$?

Answer: Yes, if S has such subset;

No, otherwise.

Example:

If S={8, 11, 16, 29, 37} and t=37,

then

- {8, 29} is a solution of Subset Sum.

- {11, 16} is a solution of Subset Sum.

- {37} is a solution of Subset Sum.

Big O Notation – Examples of NP-Complete Problems

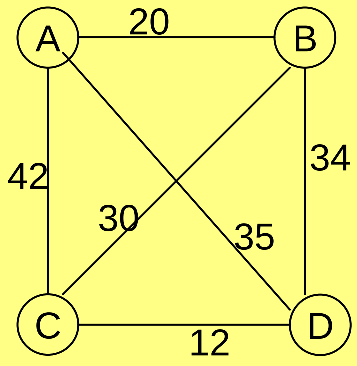
- Satisfiability
- Clique
- Hamiltonian Path
- Graph Coloring
- Subset-sum
- **Travelling Salesman**
- Scheduling

Travelling Salesman Problem:

Given a salesman

and n cities with pairwise distances between cities

Question: What is the shortest path the salesman can make such that each city is visited exactly once?



Example of 4 cities

↓

Minimal hamiltonian path in the weighted graph. (A,B,C,D)

Big O Notation – Examples of NP-Complete Problems

- ❑ Satisfiability
- ❑ Clique
- ❑ Hamiltonian Path
- ❑ Graph Coloring
- ❑ Subset-sum
- ❑ Travelling Salesman
- ❑ **Scheduling**

Scheduling Problem:

Given - a list of exams $F_1, \ldots, F_k$

- a list of students $S_1, \ldots, S_l$

- a number $h \in \mathbb{N}$

Each student is taking some specified subset of exams.

Question: Make an exam-schedule such that:

- it uses only h slots
- no student is required to take 2 exams in the same slot

Computational Limits - Undecidable Problems

There are infinitely many problems that cannot be algorithmically solved.

There are infinitely many problems that cannot be solved by computers.

Example (HALTING-PROBLEM).
There is NO ALGORITHM that decides whether a program terminates or not.