Složitost

Složitost *algoritmu* vyjadřuje nÆročnost algoritmu na různØ zdroje v průběhu výpočtu: čas výpočtu, velikost paměti, počet procesorů apod. Podle toho rozlišujeme různØ *míry složitosti*.

Míry složitosti:
- – časovÆ
- – prostorovÆ (paměťovÆ)
- – procesorovÆ (hardwarovÆ)
- ⋮

Pro definici časovØ resp. prostorovØ složitosti zavedeme pojmy *dØlka výpočtu* resp. *množství výpočtem spotřebovanØ paměti*. K přesnØ definici obou pojmů však potřebujeme tzv. *výpočetní model*. Jeho definice zÆvisí na *výpočetním paradigmatu*.

Například pro Turingův stroj dÉlkou výpočtu počet výpočetních kroků, tj. vnitřních přechodů. Pro RAM je to počet provedených instrukcí. Pro funkcionÁlní jazyk je to počet jednokrokových redukcí.

Složitost *problÉmu* je je zavedena jako složitost optimÁlního algoritmu řešícího tento problÉm. Na řešení nějakÉho problÉmu můžeme použít mnoho různých algoritmů. Časovou složitostí problÉmu pak rozumímečasovou složitost algoritmu, který řeší daný problÉm (i na „nejpomalejších datech“) nejrychleji. Prostorovou složitostí problÉmu rozumíme prostorovou složitost algoritmu, který řeší daný problÉm (i na nejmÉně příznivých datech) v nejmenší paměti apod.

Příklad:

- Řešíme problØm: seřadit vzestupně n-prvkovou posloupnost celých čísel.
- K řešení použijeme algoritmus řazení vklÆdÆním.
- Naše konkrØtní posloupnosti, kterØ budou vstupem, jsou $3, 6, 9, \ldots, 3n$ (tj. jsou už seřazenØ).

Rozlišujeme tři pojmy:

- časovou složitost problØmu
- časovou složitost algoritmu
- dØlku výpočtu

DØlka výpočtu algoritmem `InsSort` na rostoucí posloupnosti dØlky n je $6n - 4$ (tolik se provede výpočetních kroků).

ČasovÆ složitost algoritmu `InsSort` je vyjÆdřena kvadratickým polynomem $an^2 + bn + c$, jak později uvidíme. Je tedy větší než dØlka výpočtu na našich (příznivých) datech. Na jiných posloupnostech dØlky n stejný algoritmus strÆví čas více. V nejhorším případě až kvadraticky mnoho, proto je časovÆ složitost algoritmu řazení vklÆdÆním kvadratickÆ.

ČasovÆ složitost problØmu vzestupnØho seřazení posloupnosti je však lepší než kvadratickÆ: existují algoritmy, kterØ n-prvkovou posloupnost seřadí v čase $c\,n \log n$ (c je konstanta), a to i v nejhorším případě (tj. i pro „nejzlomyslněji zadanØ“ vstupní posloupnosti). Proto je časovÆ složitost problØmu řazení vyjÆdřena funkcí $c\,n \log n$ (v proměnnØ n).

DØlka výpočtu výrazů

DØlku výpočtu výrazu e označíme $\tau(e)$.

$\tau(e)$ je rovna součtu dØlek vyhodnocení všech operací, kterØ výraz e obsahuje.

Vyhodnocení elementÆrních aritmetických, logických, relačních operací nad skalÆrními operandy je konstantní. Proto i dØlku výpočtu výrazu, který obsahuje pouze tyto jednoduchØ operace, považujeme za konstantní.

Obsahuje-li však výraz e volÆní složitějších funkcí, je nutno započítat dØlku jejich výpočtu do $\tau(e)$.

Přesněji:
$\tau(c) = 0$, je-li c konstanta,
$\tau(v) = 1$, je-li c odkaz na hodnotu skalární (přepisovatelnou) proměnnou. Tedy zpřístupnění obsahu proměnnou považujeme za jednotkovou operaci.

$\tau(op\ a) = 1 + \tau(a)$, kde *op* je primitivní unární operátor, který vyhodnocuje svůj operand, například `not`.

$\tau(a \oplus b) = 1 + \tau(a) + \tau(b)$, kde $\oplus$ je některý z primitivních binárních operátorů, které vyhodnocují oba svoje operandy, například $+, -, *, /$, `div`, `mod`, $<, \leq, >, \geq, \ldots$.

Pozor, některØ operÆtory a jazykovØ konstrukce nemusí vždy vyhodnocovat všechny svoje operandy. Podle toho se pak určuje dØlka výpočtu výrazů:

$\tau(a \texttt{ \&\& } b) = 1 + \tau(a) + \tau(b)$ pro a pravdivØ
$\tau(a \texttt{ \&\& } b) = 1 + \tau(a)$ pro a nepravdivØ

$\tau(a \texttt{ || } b) = 1 + \tau(a) + \tau(b)$ pro a nepravdivØ
$\tau(a \texttt{ || } b) = 1 + \tau(a)$ pro a pravdivØ

$\tau(\texttt{if } a \texttt{ then } b \texttt{ else } c) = 1 + \tau(a) + \tau(b)$ pro a pravdivØ
$\tau(\texttt{if } a \texttt{ then } b \texttt{ else } c) = 1 + \tau(a) + \tau(c)$ pro a nepravdivØ

DØlka výpočtu volÆní funkce

Deklarace:
`f` $(x_1, \ldots, x_n)$ `=` e
VolÆní hodnotou (striktní aplikace):
$\tau(\mathtt{f}(a_1, \ldots, a_n)) = 1 + \tau(a_1) + \cdots + \tau(a_n) + \tau(e'')$, kde výraz e'' vznikne z výrazu e nahrazením každØho výskytux_i *hodnotou* výrazu a_i.

VolÆní jmØnem (normÆlní aplikace):
$\tau(\mathtt{f}(a_1, \ldots, a_n)) = 1 + \tau(e')$, kde výraz e' vznikne z výrazu e nahrazením každØho výskytu x_i *celým nevyhodnoceným výrazem* a_i.

Bude-li nÆs zajímat pouze tzv. *asymptotickØ chovÆní* složitosti algoritmů, budeme ztotožňovat složitosti lišící se jen kladnou multiplikativní konstantou. Pro takovØ případy bude užitečnÆ nÆsledující

Úmluva: Je-li e výraz obsahující jen skalÆrní konstanty a proměnnØ a elementÆrní operace (na skalÆrních hodnotÆch), klademe dØlku jeho výpočtu rovnu jednØ.

Výraz e zejmØna nesmí obsahovat vektorovØ operace (např. aritmetickØ operace na „dlouhých“ číslech) ani volÆní uživatelských funkcí.

Tuto œmluvu zavÆdíme proto, že pročœdy zjišťovÆní asymptotickØho chovÆní algoritmů je praktičtější například uvažovat dØlku přiřazení `x:=a*(b+c+d)` rovnu jednØ, než ji počítat jako $1 + 1 + 2 + 5$. Stejně víme, že součet je roven konstantě, a to nÆm stačí.

U tzv. „čistých“ výrazů předpokládáme, že nemají vedlejší efekty. To však nemusí vždy platit. Délka výpočtu výrazů, které obsahují volání funkcí, závisí nejen na samotném výrazu, ale i na *stavu*, v němž se výraz vyhodnocuje. Stavem rozumíme okamžitý obsah programových (přepisovatelných) proměnných. V jazycích s vedlejšími efekty se stav během výpočtu mění.

Příklad:

```
var a:Integer;
function f (n:Integer):Integer;
 var i,k:Integer;
 begin k := 0;
       for i:=0 to n*a do
           k := k+i;
       a := k;
       f:=a  {return a}
 end
```

GlobÆlní přepisovatelnÆ proměnnÆ a mÆ na začÆtku hodnotu 2, pak ve výrazu `f(4)+f(4)` trvÆ každØ vyhodnocení stejnØho podvýrazu `f(4)` různou dobu. (Jakou?)

DØlka výpočtu příkazů

Nechť S je množina stavů, $\sigma \in S$. Každý příkaz p funguje jako *stavový transformÆtor* $\nu(p) : S \dashrightarrow S$.

DØlku výpočtu příkazu p ve stavu σ označíme $\tau(p, \sigma)$.

PrÆzdný příkaz

$\tau(\texttt{skip}, \sigma) = 0$ pro libovolný stav σ

Přiřazovací příkaz

$\tau(\texttt{v:=}e, \sigma) = 1 + \tau(e, \sigma)$, je-li v jednoduchÆ přepisovatelnÆ proměnnÆ

Sekvence

$$\tau(\texttt{begin}\, p_1\,\texttt{;}\,\ldots\,\texttt{;}\, p_k\,\texttt{end}, \sigma_0) = \sum_{i=1}^{k} \tau(p_i, \sigma_{i-1}), \quad \text{kde } \sigma_i = \nu(p_i)(\sigma_{i-1})$$

Větvení

Nechť σ je stav, p, q příkazy, e výraz a $\sigma' = \nu(e)(\sigma)$.

$\tau(\texttt{if}\ e\ \texttt{then}\ p\ \texttt{else}\ q, \sigma) = \tau(e, \sigma) + \tau(p, \sigma')$ pro pravdivØ e,

$\tau(\texttt{if}\ e\ \texttt{then}\ p\ \texttt{else}\ q, \sigma) = \tau(e, \sigma) + \tau(q, \sigma')$ pro nepravdivØ e.

Pokud výraz e nemÆ vedlejší efekty, pak $\sigma = \sigma'$ a

$\tau(\texttt{if}\ e\ \texttt{then}\ p\ \texttt{else}\ q, \sigma) = \tau(e, \sigma) + \tau(p, \sigma)$ pro pravdivØ e,

$\tau(\texttt{if}\ e\ \texttt{then}\ p\ \texttt{else}\ q, \sigma) = \tau(e, \sigma) + \tau(q, \sigma)$ pro nepravdivØ e.

Cyklus while

Nechť $t = (\texttt{while}\ \ e\ \ \texttt{do}\ \ s)$ je příkaz cyklu a σ_n je stav takový, že ze stavu σ_0 příkaz cyklu t zopakuje právě n-krát svého tělo s.

Pro $0 \leq i \leq n$ označme $\sigma'_i = \nu(e)(\sigma_i)$,

pro $0 \leq i < n$ označme $\sigma_{i+1} = \nu(s)(\sigma_i)$,

a nechť pro $0 \leq i < n$ je $\mu(e)(\sigma_i) =$ pravda, $\mu(e)(\sigma_n) =$ nepravda. Pak

$$\tau(t, \sigma_0) = \sum_{0 \leq i < n} \big(\tau(e, \sigma_i) + \tau(s, \sigma'_i)\big) + \tau(e, \sigma_n)$$

Pokud výraz e nemá vedlejší efekty, pak $\sigma_i = \sigma'_i$ a

$$\tau(t, \sigma_0) = \sum_{0 \leq i < n} \big(\tau(e, \sigma_i) + \tau(s, \sigma_i)\big) + \tau(e, \sigma_n)$$

$\hat{T}$ značí množinu všech hodnot typu T, tzv. *domØnu*

StejnØ rozšíření definice dØlky výpočtu se vztahuje i na výrazy s vedlejšími efekty.

Příkazy můžeme považovat za výrazy, kterØ mají speciÆlní hodnotu $\bullet$ typu $\mathtt{Com}$. SØmantika každØho příkazu je pak konstantní funkce, tj. pro libovolný příkaz p a stav σ je $\mu(p)(\sigma) = \bullet$.

Nechť `f` je funkce definovanÆ

`f (`x_1`,...,`x_n`) =` e

a σ_0 je stav, ve kterØm začne výpočet.

Pak buďto, v případě volÆní jmØnem,

$\tau(\texttt{f}(a_1,\ldots,a_n),\sigma_0) = 1 + \tau(e',\sigma_0)$, kde výraz e' vznikne z výrazu e (těla funkce `f`) nahrazením každØho výskytux_i *výrazem* a_i

anebo, v případě volÆní hodnotou,

$\tau(\texttt{f}(a_1,\ldots,a_n),\sigma_0) = 1 + \tau(a_1,\sigma_0) + \tau(a_2,\sigma_1) + \cdots + \tau(a_n,\sigma_{n-1}) + \tau(e'',\sigma_n)$,

kde $\sigma_i = \nu(a_i)(\sigma_{i-1})$ (pro $0 < i \leq n$) a výraz e'' vznikne z těla e takto:

$$
\begin{array}{lll}
e'' = & \texttt{begin} & x_1 \texttt{:=} \mu(a_1)(\sigma_0)\texttt{;} \\
 & & \vdots \\
 & & x_n \texttt{:=} \mu(a_n)(\sigma_{n-1})\texttt{;} \\
 & & e \\
 & \texttt{end} &
\end{array}
$$

DØlku výpočtu příkazu cyklu for můžeme odvodit tak, že si cyklus for vyjÆdříme pomocí cyklu while.

Nechť $t = (\texttt{for i:=}e\ \texttt{to}\ e'\ \texttt{do}\ s)$ je příkaz cyklu a σ je stav takový, že ze stavu σ příkaz cyklu t zopakuje prÆvě n-krÆt svØtělo s.

Označíme $\sigma_0 = \nu(\texttt{i:=}e)(\sigma)$,
pro $0 \leq i \leq n$ označíme $\sigma'_i = \nu(\texttt{i} \leq e')(\sigma_i)$,
a pro $0 \leq i < n$ označíme $\sigma''_i = \nu(s)(\sigma'_i)$, $\sigma_{i+1} = \nu(\texttt{i:=succ(i)})(\sigma''_i)$

Pak

$$\tau(t,\sigma) = \tau(\texttt{i:=}e)(\sigma) + \sum_{0 \leq i < n} (\tau(\texttt{i} \leq e', \sigma_i) + \tau(s, \sigma'_i) + \tau(\texttt{i:=succ(i)}, \sigma''_i)) + \tau(\texttt{i} \leq e', \sigma_n)$$

Většinou nÆs víc než dØlka jednoho konkrØtního výpočtu zajímÆ, jak se daný algoritmus chovÆ obecně, tj. jak vypadÆ množina dØlek všech možných výpočtů — pro množinu všech přípustných vstupních dat (vyhovujících vstupní podmínce).

ČasovÆ složitost algoritmu

Je-li *In* množina všech vstupních dat pro algoritmus A, pak pro $x \in$ *In* zavedeme číslo $|x| \in \mathbb{N}$, kterØ nazveme *velikost* vstupní hodnoty x.

PočÆteční stav výpočtu, který odpovídÆ umístění hodnoty x na vstupu, označíme σ_x.

Časovou složitost algoritmu A zavedeme jako jako funkci $T_A : \mathbb{N} \to \mathbb{N}$ takto:

$$T_A(n) = \max\{\tau(A, \sigma_x) \mid |x| = n\}$$

ČasovÆ složitost algoritmu je tedy funkce, kterÆ pro každou velikost vstupních dat je rovna dØlce *nejdelšího* výpočtu na všech možných datech tØto velikosti.

Velikost vstupních dat odpovídÆ počtu bitů, kterými je vstupní œdaj vyjÆřen. V praxi mÆ často tyto významy:

číslo n	dØlka jeho bitovØho zÆpisu, tj. $\lceil \log_2 n \rceil$
posloupnost čísel	počet jejích prvků
graf	počet uzlů $+$ počet hran

Protože množina v definici složitosti je většinou nekonečnÆ, musíme hledanØ maximum určit *analýzou nejhoršího případu*.

V některých případech je výhodnější rozdělit velikost dat na dvě čísla (graf) a upravit definici složitosti $T_A : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$

Příklad: Algoritmus řazení vklÆdÆním (imperativní verze – Pascal)

```
1   const MAX = 999;
2   type Elem = Integer;
3        Posl = array [1..MAX] of Elem;
4   procedure iInsSort (n:Integer; var A:Posl);
5     var i, j : Integer;  x : Elem;
6     begin
7         for i := 2 to n do
8         begin
9             x := A[i];  j := i-1;
10            while (j>0) && (A[j]>x) do
11            begin
12                A[j+1] := A[j];
13                j := j-1
14            end;
15            A[j+1] := x
16         end
17     end
```

Příklad: Analýza nejhoršího případu

Algoritmus: `iInsSort`
Vstup: posloupnost celých čísel $A = (a_1, \ldots, a_n)$
dØlky n.
Výstup: posloupnost celých čísel B, kterÆ je permutací posloupnosti A a přitom je neklesající.

Označme c_i dØlku výpočtu příkazu nebo testu na i-tØmřÆdku algoritmu `iInsSort`, přičemž na sedmØmřÆdku (v zÆhlaví cyklu for) jsou uvedeny elementÆrní akce $c_{7,\text{init}}$, $c_{7,\text{test}}$, $c_{7,\text{inc}}$.

Pak dØlka výpočtu je

$$T(n) = c_{7,\text{init}} + \sum_{i=2}^{n} \left(c_{7,\text{test}} + c_9 + \sum_{j=\xi}^{i-1} (c_{10} + c_{12} + c_{13}) + c_{10} + c_{15} + c_{7,\text{inc}} \right) + c_{7,\text{test}}$$

kde ξ je hodnota, při kterØ se vnitřní cyklus algoritmu zopakuje nejvícekrÆt, tj. nejmenší možnÆ hodnota j, při níž je ještě splněna podmínka za while (na 10. řÆdku algoritmu).

TakovØ minimÆlní možnØ ξ je zřejmě rovno 1, a to v každØm průchodu vnějším cyklem for. Situace, kdy v každØm průchodu vnějším cyklem je $\xi = 1$, nastane, když vstupní posloupnost A je klesající.

Po dosazení $\xi = 1$ a po úpravě dostáváme

$$T(n) = \frac{c_{10} + c_{12} + c_{13}}{2} n^2 + (c_{7,\text{test}} + c_{7,\text{inc}} + c_9 + \frac{c_{10}}{2} - \frac{c_{12}}{2} - \frac{c_{13}}{2} + c_{15}) n + (c_{7,\text{init}} - c_{7,\text{inc}} - c_9 - c_{10} - c_{15})$$

což je kvadratický polynom v proměnné n.

Podle dřívější úmluvy lze všechny konstanty považovat za jedničky (resp. $c_9 = 2$), takže po dosazení lze pracovat s polynomem $\frac{3}{2} n^2 + \frac{9}{2} n - 4$.

Abychom mohli při porovnÆvÆní složitostí algoritmů abstrahovat od rychlosti konkrØtního počítače, budeme chtít srovnÆvat funkce podle toho, „jak rychle rostou“, přičemž za významný rozdíl v rychlosti růstu nepovažujeme například případ, kdy jedna funkce mÆ c-krÆt větší hodnoty než druhÆ (pro nějakou kladnou konstantu c).

Rychlost růstu funkcí

Nechť $g : \mathbb{N} \to \mathbb{N}$. Zavedeme množiny funkcí:

Množina funkcí rostoucích nejvýše tak rychle jako g :

$$O(g) = \{f \mid \exists c > 0 \exists n_0 \forall n \geq n_0 . \, 0 \leq f(n) \leq c \cdot g(n)\}$$

Množina funkcí rostoucích aspoň tak rychle jako g :

$$\Omega(g) = \{f \mid \exists c > 0 \exists n_0 \forall n \geq n_0 . \, 0 \leq f(n) \geq c \cdot g(n)\}$$

Množina funkcí rostoucích stejně rychle jako g :

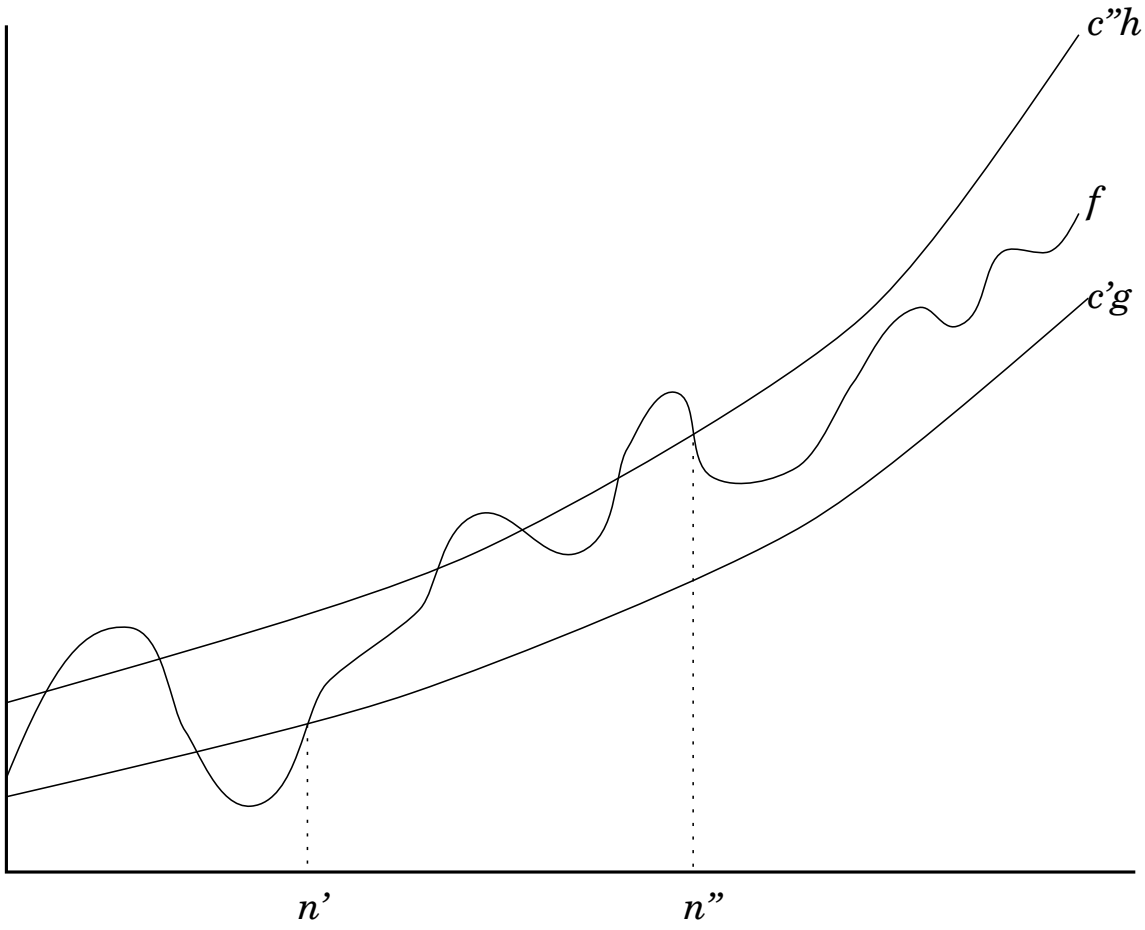
$$\Theta(g) = O(g) \cap \Omega(g)$$

Množina funkcí rostoucích pomaleji než g :

$$o(g) = \{f \mid \forall c > 0 \exists n_0 \forall n \geq n_0 . \, 0 \leq f(n) < c \cdot g(n)\}$$

Množina funkcí rostoucích rychleji než g :

$$\omega(g) = \{f \mid \forall c > 0 \exists n_0 \forall n \geq n_0 . \, 0 \leq c \cdot g(n) < f(n)\}$$



$$f \in \Omega(g) \Leftrightarrow \forall n \geq n'.\, 0 \leq c'g(n) \leq f(n)$$

$$f \in O(h) \Leftrightarrow \forall n \geq n''.\, f(n) \leq c''h(n)$$

Protože se budeme zabývat především funkcemi vyjadřujícími složitost algoritmu (kterÆ je vždy nezÆpornÆ), omezíme se dÆle na nezÆpornØ funkce.

Vlastnosti množin O, Ω, Θ, o, ω

Věta: Jsou-li $f, g : \mathbb{N} \to \mathbb{N}$ dvě funkce, pak $f \in \Theta(g)$ právě když

$$\exists c_1 > 0 \exists c_2 > 0 \exists n_0 \forall n \geq n_0 .\ c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

Věta: Pro každé dvě kladné funkce $f, g : \mathbb{N} \to \mathbb{N}$ platí:

$$f \in O(g) \Leftrightarrow g \in \Omega(f)$$
$$f \in o(g) \Leftrightarrow g \in \omega(f)$$
$$f \in \Theta(g) \Leftrightarrow g \in \Theta(f)$$
$$f \in o(g) \Rightarrow f \in O(g)$$
$$f \in \omega(g) \Rightarrow f \in \Omega(g)$$

V důkazu prvních dvou tvrzení stačí obrÆtit nerovnost a uvÆžit kladnou konstantu $\frac{1}{c}$.

Třetí tvrzení je triviÆlní.

ČtvrtØ tvrzeníříkÆ, že jistÆ vlastnost platí pro všechna kladnÆ. Ale kladnÆčísla existují, například $c = 1$. Tedy platí takØ slabší vlastnost pro *nějakØ* $c > 0$.

Důkaz pÆtØho tvrzení je analogický.

Věta: Pro každØ dvě kladnØ funkce $f, g : \mathbb{N} \to \mathbb{N}$ platí:

$$f \in o(g) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

$$f \in \omega(g) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

Rovnost $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ je ekvivalentní s podmínkou $\forall c > 0 \exists n_0 \forall n \geq n_0 . \left| \frac{f(n)}{g(n)} \right| < c$. Obě funkce jsou kladnØ, tedy absolutní hodnotu nemusíme uvažovat a nerovnost lze vynÆsobit číslem $g(n)$, čímž dostaneme podmínku pro $f \in o(g)$. Úprava byla ekvivalentní, takže ekvivalentní jsou i obě podmínky v prvním tvrzení.

Rovnost $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ je ekvivalentní s podmínkou $\forall c > 0 \exists n_0 \forall n \geq n_0 . \left| \frac{f(n)}{g(n)} \right| > c$. Obě funkce jsou kladnØ, tedy absolutní hodnotu nemusíme uvažovat a nerovnost lze vynÆsobit číslem $g(n)$, čímž dostaneme podmínku pro $f \in \omega(g)$. Úprava byla ekvivalentní, takže ekvivalentní jsou i obě podmínky v prvním tvrzení.

Jsou-li $f, g : \mathbb{N} \to \mathbb{N}$ dvě kladnØ funkce, pak platí

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \;\Rightarrow\; f \in O(g)$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0 \;\Rightarrow\; f \in \Omega(g)$$

ObrÆcenØ implikace však obecně neplatí, protože uvedenØ limity nemusí existovat.

Pro funkce f, g však platí silnější tvrzení:

$$\limsup_{n\to\infty} \frac{f(n)}{g(n)} < \infty \;\Leftrightarrow\; f \in O(g)$$

$$\liminf_{n\to\infty} \frac{f(n)}{g(n)} > 0 \;\Leftrightarrow\; f \in \Omega(g)$$

Cvičení: Rozhodněte a zdůvodněte, zda pro každou kladnou funkci g platí
$O(g) = o(g) \cup \Theta(g),\ \Omega(g) = \omega(g) \cup \Theta(g)$.

Věta: Nechť $f, g, h : \mathbb{N} \to \mathbb{N}$ jsou tři kladnØ funkce a nechť funkce $q : \mathbb{N} \to \mathbb{N}$ je definovÆna $q(n) = f(n) + g(n)$.

Pak platí:

$$f \in O(h) \wedge g \in O(h) \Rightarrow q \in O(h)$$

$$f \in \Omega(h) \wedge g \in \Omega(h) \Rightarrow q \in \Omega(h)$$

$$f \in \Theta(h) \wedge g \in \Theta(h) \Rightarrow q \in \Theta(h)$$

$$f \in o(h) \wedge g \in o(h) \Rightarrow q \in o(h)$$

$$f \in \omega(h) \wedge g \in \omega(h) \Rightarrow q \in \omega(h)$$

PoznÆmka: Platí řada podobných odvozených vlastností, například

$$f \in \Omega(h) \wedge g \in O(h) \Rightarrow q \in \Omega(h)$$
$$f \in \Theta(h) \wedge g \in O(h) \Rightarrow q \in \Theta(h)$$
$$f \in \Theta(h) \wedge g \in o(h) \Rightarrow q \in \Theta(h)$$

apod.

Cvičení: Formulujte a dokažte další vlastnosti, kterØ z věty vyplývají.

Věta: Nechť $f, h : \mathbb{N} \to \mathbb{N}$ jsou dvě kladnØ funkce, c, d konstanty, $c > 0$, a nechť funkce $p : \mathbb{N} \to \mathbb{N}$ je definovÆna $p(n) = c \cdot f(n) + d$.

Pak platí:

$$f \in O(h) \Rightarrow p \in O(h)$$

$$f \in \Omega(h) \Rightarrow p \in \Omega(h)$$

$$f \in \Theta(h) \Rightarrow p \in \Theta(h)$$

$$f \in o(h) \Rightarrow p \in o(h)$$

$$f \in \omega(h) \Rightarrow p \in \omega(h)$$

Důkaz: Nejprve nechť $d = 0$; tvrzení obdržíme vhodnou volbou kladnØ konstanty v definiích tříd O, Ω, Θ, o, ω. Pro obecnØ $d \neq 0$ jde o speciÆlní případ předchozí věty, kdy funkce g z jejích předpokladů je konstantní.

Věta: Nechť $f, g, h : \mathbb{N} \to \mathbb{N}$ jsou tři funkce, pak

$$f \in O(g) \wedge g \in O(h) \Rightarrow f \in O(h)$$
$$f \in o(g) \wedge g \in O(h) \Rightarrow f \in o(h)$$
$$f \in O(g) \wedge g \in o(h) \Rightarrow f \in o(h)$$
$$f \in \Omega(g) \wedge g \in \Omega(h) \Rightarrow f \in \Omega(h)$$
$$f \in \omega(g) \wedge g \in \Omega(h) \Rightarrow f \in \omega(h)$$
$$f \in \Omega(g) \wedge g \in \omega(h) \Rightarrow f \in \omega(h)$$

Důsledek: Relace *„růst nejvýše tak rychle“* tvoří *předuspořádání* na množině všech funkcí $\mathbb{N} \to \mathbb{N}$.

Poznámka: Relace *„růst právě tak rychle“* tvoří *ekvivalenci* na množině všech funkcí $\mathbb{N} \to \mathbb{N}$.

Tyto relace však nejsou úplné. Existují dvojice funkcí, které nejsou porovnatelné.

PoznÆmka: *PředuspořÆdÆní* (též *polouspořÆdÆní*) je binÆrní relace, kterÆ je reflexivní a transitivní. *UspořÆdÆní* je binÆrní relace, kterÆ je reflexivní, antisymetrickÆ a transitivní. *Ekvivalence* je binÆrní relace, kterÆ je reflexivní, symetrickÆ a transitivní.

Cvičení: Najděte příklad dvojice funkcí $f, g : \mathbb{N} \to \mathbb{N}$, kterØ nejsou porovnatelnØ, tj. $f \notin O(g), f \notin \Omega(g)$.

Cvičení: Dokažte tvrzení:
Je-li kladnÆ funkce f skoro všude majorizovÆna funkcí g (tj. $f(n) \leq g(n)$ pro skoro všechna n), pak $f \in O(g)$.

Cvičení: Dokažte tvrzení:
Pro každou kladnou funkci g je $o(g) \cap \omega(g) = \emptyset$.

Růst jednoduchých funkcí

Def: *Kladný polynom* je polynom s kladným vedoucím koeficientem.

Věta: Kladný polynom vyššího stupně roste vždy rychleji než polynom nižšího stupně. Dva kladnØ polynomy stejnØho stupně rostou stejně rychle.

PoznÆmka: Předchozí větu lze zobecnit i na mocninnØ funkce s neceločíselnými exponenty: jestliže $0 \leq a < b$, pak $n^a \in o(n^b)$.

Věta: ExponenciÆlní funkce se zÆkladem $a > 1$ roste rychleji než libovolný polynom. LogaritmickØ funkce rostou pomaleji než funkce lineÆrní.

Cvičení: Dokažte, že pro každé přirozené číslo k platí $(n+1)^k \in O(n^k)$.
(Vhodnou kladnou konstantu c z definice množiny O určete podle binomické věty.)

Věta: Nechť $1 < a < b$. Pak

$$\log_a \in \Theta(\log_b)$$

$$a^n \in o(b^n)$$

PoznÆmka: Roste-li funkce logaritmicky, pak na zÆkladu logaritmu nezÆleží. (Jen musí být větší než 1, což je však nutnÆ podmínka k tomu, aby funkce vůbec rostla.)

Věta: Platí $n! \in O(n^n)$, $n! \in \Omega(2^n)$.

Důkaz plyne z definice faktoriÆlu.

PoznÆmka: Místo „$\Theta(\log)$“ se často píše „$\Theta(\log\, n)$“, podobně jako „$o(n^2)$“ místo „$o(f)$, kde $f(n) = n^2$ pro každé n“,
a dokonce „$O(1)$“ místo „$O(g)$, kde $g(n) = c > 0$“.
Mnozí autoři zachÆzejí ve zneužívÆní notace ještě dÆle (např. pracují s množinami funkcí, jako by to byla čísla).

PoznÆmka: Z tzv. Stirlingovy aproximace

$$\sqrt{2\pi n}(n/\mathrm{e})^n \le n! \le \sqrt{2\pi n}(n/\mathrm{e})^n \mathrm{e}^{1/(12n)}$$

vyplývají silnější tvrzení o růstu faktoriÆlu:

$$n! \in o(n^n), \qquad n! \in \omega(2^n)$$

Def: Řekneme, že funkce f *roste nejvýše polylogaritmicky*, když existuje číslo $k > 0$ tak, že $f \in O\left((\log n)^k\right)$.

Def: Řekneme, že funkce f *roste nejvýše polynomiálně*, když existuje číslo $k > 0$ tak, že $f \in O(n^k)$.

Funkce f roste *roste aspoň polynomiálně*, když existuje číslo $\varepsilon > 0$ tak, že $f \in \Omega(n^\varepsilon)$.

Def: Řekneme, že funkce f *roste nejvýše exponenciálně*, když existuje číslo $a > 1$ tak, že $f \in O(a^n)$.

Funkce f *roste aspoň exponenciálně*, když existuje číslo $a > 1$ tak, že $f \in \Omega(a^n)$.

Def: Řekneme, že funkce f *roste aspoň dvojitě exponenciálně*, když existuje číslo $a > 1$ tak, že $f \in \Omega\left(a^{a^n}\right)$.

PoznÆmka: Existují funkce, kterØ rostou rychleji než všechny k-nÆsobnØ exponenciÆlní funkce, a to dokonce pro libovolnØ k.
Naopak existují kladnØ funkce, kterØ rostou (tedy rychlejinež konstantní), ale rostou pomaleji než funkce $\log \circ \cdots \circ \log$, kde logaritmus můžeme složit sama se sebou libovolněkrÆt.

Def: *Fibonacciho funkce* $F : \mathbb{N} \to \mathbb{N}$ je definovaná rekursivně:

$$\begin{aligned} F(0) &= 0 \\ F(1) &= 1 \\ F(n+2) &= F(n) + F(n+1) \end{aligned}$$

pro všechna $n \in \mathbb{N}$.

Věta: Nechť $\varphi = \frac{1+\sqrt{5}}{2}$, $\hat{\varphi} = \frac{1-\sqrt{5}}{2}$. Pak pro každé n platí $F(n) = \frac{\varphi^n - \hat{\varphi}^n}{\sqrt{5}}$.

Důsledek: Pro Fibonacciho funkci F platí $F \in \Theta(\varphi^n)$, kde $\varphi = \frac{1+\sqrt{5}}{2}$.

Důkaz Věty – indukcí podle n.

(i) Pro $n = 0$ a $n = 1$ tvrzení platí:

$$F(0) = \frac{\varphi^0 - \hat{\varphi}^0}{\sqrt{5}} = \frac{1-1}{\sqrt{5}} = 0, \quad F(1) = \frac{\varphi^1 - \hat{\varphi}^1}{\sqrt{5}} = \frac{\frac{1+\sqrt{5}}{2} - \frac{1-\sqrt{5}}{2}}{\sqrt{5}} = 1$$

(ii) Předpokládejme, že pro nějaké $n \geq 0$ platí $F(n) = \frac{\varphi^n - \hat{\varphi}^n}{\sqrt{5}}$,

$F(n+1) = \frac{\varphi^{n+1} - \hat{\varphi}^{n+1}}{\sqrt{5}}$, a ukážeme, že $F(n+2) = \frac{\varphi^{n+2} - \hat{\varphi}^{n+2}}{\sqrt{5}}$.

$$F(n+2) = F(n) + F(n+1) = \frac{\varphi^n - \hat{\varphi}^n}{\sqrt{5}} + \frac{\varphi^{n+1} - \hat{\varphi}^{n+1}}{\sqrt{5}} =$$

$$\frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n + \left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}}{\sqrt{5}} =$$

$$\frac{4\left(1+\sqrt{5}\right)^n - 4\left(1-\sqrt{5}\right)^n + 2\left(1+\sqrt{5}\right)^{n+1} - 2\left(1-\sqrt{5}\right)^{n+1}}{2^{n+2}\sqrt{5}} =$$

$$\frac{\left(4+2(1+\sqrt{5})\right)\left(1+\sqrt{5}\right)^n-\left(4+2(1-\sqrt{5})\right)\left(1-\sqrt{5}\right)^n}{2^{n+2}\sqrt{5}}=$$
$$\frac{\left(1+2\sqrt{5}+5\right)\left(1+\sqrt{5}\right)^n-\left(1-2\sqrt{5}+5\right)\left(1-\sqrt{5}\right)^n}{2^{n+2}\sqrt{5}}=$$
$$\frac{\left(1+\sqrt{5}\right)^{n+2}-\left(1-\sqrt{5}\right)^{n+2}}{2^{n+2}\sqrt{5}}=\frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n+2}-\left(\frac{1-\sqrt{5}}{2}\right)^{n+2}}{\sqrt{5}}=\frac{\varphi^{n+2}-\hat{\varphi}^{n+2}}{\sqrt{5}}$$

Lemma: Funkce $\log(n!)$ roste stejně rychle jako funkce $n \cdot \log n$.

Důkaz: Nejdříve ukÆžeme, že $\log_2 n! \in O(n \log n)$.

$\log_2 n! = \sum_{k=1}^{n} \log_2 k \leq \sum_{k=1}^{n} \log_2 n = n \log n$, takže $\log_2 n! \in O(n \log n)$.

DÆle funkci $\log_2 n!$ ohraničíme i zdola: $2 \log_2 n! = 2 \sum_{k=1}^{n} \log_2 k \geq 2 \sum_{k=\lfloor n/2 \rfloor}^{n} \log_2 k \geq$

$2 \sum_{k=\lfloor n/2 \rfloor}^{n} \log_2 \left\lfloor \frac{n}{2} \right\rfloor = 2 \left\lfloor \frac{n}{2} \right\rfloor \log_2 \left\lfloor \frac{n}{2} \right\rfloor \in \Theta(n \log n)$, takže $\log_2 n! \in \Omega(n \log n)$.

Dohromady $\log n! \in \Theta(n \log n)$.

Cvičení: Dokažte, že dØlky výpočtů podle nějakØho algoritmu A jsou v množině $\Theta(g)$, prÆvě když časovÆ složitost algoritmu A je v $O(g)$ a dØlky nejkratších výpočtů patří do $\Omega(g)$.

Cvičení: Seřaďte podle rychlosti růstu funkce (v proměnnØ n): log n^n, log(log n), $n\sqrt{n}$, $2^{\log_3 n}$.