Výpočetní složitost algoritmů

Složitost algoritmu

- Počítače pracují rychle, ale ne nekonečně rychle. Provedení každé instrukce trvá nějakou (i když velmi krátkou) dobu.
- Stejný problém může řešit více různých algoritmů a doba výpočtu může být různá — chtěli bychom mít možnost algoritmy vzájemně porovnat.
- Algoritmy můžeme naprogramovat a změřit čas výpočtu. Tím zjistíme jak dlouho trvá výpočet na konkrétních datech, na kterých algoritmus testujeme.
- Chtěli bychom mít i nějakou přesnější představu o tom, jak dlouho bude trvat výpočet na všech možných vstupních datech.

Složitost algoritmu

- Doba výpočtu je ovlivněna mnoha faktory, např.:
  - použitý algoritmus
  - množství vstupních dat
  - použitý hardware (důležitá může být např. taktovací frekvence procesoru)
  - použitý programovací jazyk — a jeho konkrétní implementace (překladač/interpreter)
  - ...
- Pokud potřebujeme řešit problém pro „malá“ vstupní data, doba výpočtu je většinou zanedbatelná.
- S narůstajícím množstvím vstupních dat (velikosti vstupu) může doba výpočtu růst, někdy velmi výrazně.

Složitost algoritmu

- **Časová složitost algoritmu** — jak závisí doba výpočtu na množství vstupních dat
- **Paměťová** (resp. **prostorová**) **složitost algoritmu** — jak závisí množství použité paměti na množství vstupních dat

Poznámka: Přesné definice budou uvedeny později.

Poznámka:

- Existují i další typy výpočetní složitosti, kterými se nebudeme zabývat (např. komunikační složitost).

Složitost algoritmu

Přesné určení doby výpočtu nebo množství použité paměti může být extrémně komplikované.

Většinou se při analýze výpočetní složitosti algoritmu používá celá řada zjednodušení:

- Většinou se neanalyzuje, jak závisí doba výpočtu nebo množství použité paměti na konkrétních vstupních datech, ale pouze, jak závistí na **velikosti vstupu**, tj. na množství těchto dat.

- Funkce vyjadřující, jak roste doba výpočtu nebo množství použité paměti v závislosti na velikosti vstupu, se nepočítají přesně — počítají se **odhady** těchto funkcí.

- Odhady těchto funkcí se vyjadřují pomocí tzv. **asymptotické notace** — např. se řekne, že časová složitost algoritmu MergeSort je $O(n \log n)$, zatímco časová složitost algoritmu BubbleSort je $O(n^2)$.

Velikost vstupu

Velikost vstupu — hodnota udávající, jak je daná vstupní instance „velká“

- Nejčastěji je velikost vyjádřena jako jediné číslo — obvykle se označuje toto číslo n nebo N.
- Někdy je vhodnější vyjádřit velikost dvojicí (občas i trojicí, čtveřicí, atd.) parametrů — v tom případě se často označují n a m (nebo N a M).
- Co přesně bude považováno za velikost vstupu, si můžeme zvolit.

Velikost vstupu

Příklady toho, co například může být velikostí vstupu:

- Vstupem je sekvence nějakých hodnot, pole prvků apod. (např. u problému třídění, vyhledávání v poli, hledání maximálního prvku, apod.):

  n — počet prvků v této sekvenci nebo poli

- Vstupem je řetězec znaků (slovo z nějaké abecedy):

  n — počet znaků v tomto řetězci

- Vstupem jsou dva řetězce, např. (dlouhý) text, který se bude prohledávat, a (kratší) hledaný řetězec:

  n — počet znaků v prohledávaném textu
  m — počet znaků hledaného řetězce

Velikost vstupu

- Vstupem je množina řetězců:

  Jedna možnost:

  n — součet délek všech řetězců

  Jiná varianta:

  n — součet délek všech řetězců, m — počet řetězců

- Vstupem je graf:

  n — počet vrcholů, m — počet hran

Velikost vstupu

- Vstupem je jedno číslo (např. u testování prvočíselnosti):

  Jedna možnost:

  n — počet bitů daného čísla — např. velikost vstupu 962261 je 20

  Jiná varianta:

  n — hodnota daného čísla — velikost vstupu 962261 je 962261

- Vstupem je posloupnost čísel, přičemž hodnoty těchto čísel ovlivňují dobu výpočtu (např. u problému, kde je cílem spočítat největšího společného dělitele všech čísel v dané posloupnosti):

  n — součet počtu bitů všech čísel v dané posloupnosti

Doba výpočtu

Řekněme, že máme:

- algoritmus Alg řešící problém P (resp. konkrétní implementaci algoritmu Alg),
- stroj $\mathcal{M}$ vykonávající algoritmus Alg,
- vstup w z množiny In, což je množina všech vstupů pro problém P

Příklad:

- konkrétní implementace algoritmu Quicksort v jazyce C++ řešící problém třídění,
- počítač s daným konkrétním typem procesoru, s určitou konkrétní frekvencí, na které pracuje procesor, s daným konkrétním množstvím paměti, operačním systémem, atd.
- vstup: pole $[6, 13, 1, 8, 4, 5, 8]$
  (pozn.: realističtější by byl příklad pole s milionem prvků)

Doba výpočtu

$t(w)$ — doba výpočtu algoritmu Alg nad vstupem w na stroji $\mathcal{M}$

V jakých jednotkách dobu výpočtu udávat? (Uvidíme, že ve skutečnosti to při použití asymptické notace není podstatné.)

- **v sekundách** — závisí na příliš mnoha detailech implementace, těžké určit jinak než měřením

  (i na tomtéž počítači s těmi samými daty může doba výpočtů různě kolísat)

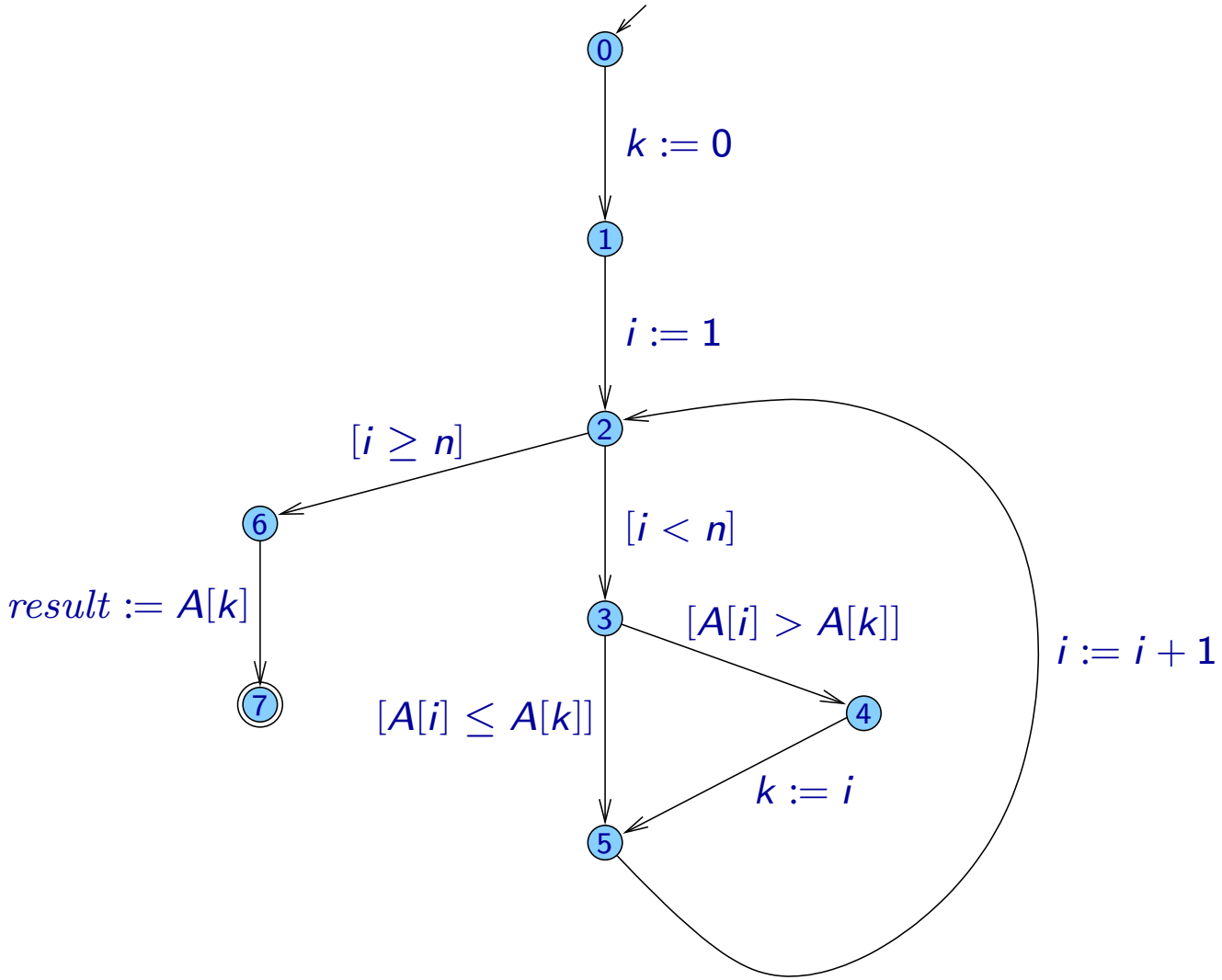
- **v počtu provedených kroků** — je třeba specifikovat, co považovat za jedek krok, například:
  - jeden příkaz vyššího programovacího jazyka
  - jedna instrukce strojového kódu nebo bytekódu
  - jeden takt procesoru
  - jedna operace určitého typu — např. porovnání, aritmetická operace, apod. (přičemž ostatní operace jsou ignorovány)
  - ...

Doba výpočtu

Řekněme, že máme algoritmus reprezentován ve formě grafu řídícího toku:

- Každé instrukci (tj. každé hraně) přiřadíme hodnotu udávající, jak dlouho trvá provedení této instrukce.
- Provedení různých instrukcí může trvat různou dobu.
- Pro jednoduchost předpokládejme, že provedení té samé instrukce trvá pokaždé stejnou dobu — hodnota přiřazená dané instrukci je číslo z množiny $\mathbb{R}^+$ (množina nezáporných reálných čísel).

Doba výpočtu



Instr.	doba
$k := 0$	c_0
$i := 1$	c_1
$[i < n]$	c_2
$[i \geq n]$	c_3
$[A[i] \leq A[k]]$	c_4
$[A[i] > A[k]]$	c_5
$k := i$	c_6
$i := i + 1$	c_7
$result := A[k]$	c_8

Doba výpočtu

Příklad: Doby provedení jednotlivých instrukcí by mohly být třeba:

Instr.	označení	doba
$k := 0$	c_0	4
$i := 1$	c_1	4
$[i < n]$	c_2	10
$[i \geq n]$	c_3	12
$[A[i] \leq A[k]]$	c_4	14
$[A[i] > A[k]]$	c_5	12
$k := i$	c_6	5
$i := i + 1$	c_7	6
$result := A[k]$	c_8	5

Pro konkrétní vstup w, např. pro $w = ([3, 8, 4, 5, 2], 5)$, bychom mohli výpočet odsimulovat a určit konkrétní dobu výpočtu $t(w)$.

Časová složitost algoritmu

Řekněme, že:

- Pro daný algoritmus Alg a stroj $\mathcal{M}$ a každý vstup w z množiny všech vstupů In je přesně definována doba výpočtu $t(w)$.
- Každému vstupu w z množiny In je přiřazeno číslo $size(w)$ udávající velikost vstupu w.

  (Formálně se jedná o funkci $size : In \to \mathbb{N}$.)

Definice

Časová složitost algoritmu Alg v nejhorším případě je funkce $T : \mathbb{N} \to \mathbb{R}^+$, která každému přirozenému číslu n přiřazuje maximální dobu výpočtu algoritmu Alg nad vstupem velikosti n.
Pro každé $n \in \mathbb{N}$ tedy platí:

- Pro každý vstup $w \in In$ takový, že $size(w) = n$, je $t(w) \leq T(n)$.
- Existuje vstup $w \in In$ takový, že $size(w) = n$ a $t(w) = T(n)$.

Časová složitost algoritmu

Z definice vidíme, že časová složitost algoritmu je funkce, jejíž přesné hodnoty závistí nejen na daném algoritmu Alg, ale také na následujících věcech:

- na stroji $\mathcal{M}$, na kterém algoritmus Alg běží,
- na definici doby výpočtu $t(w)$ algoritmu Alg na stroji $\mathcal{M}$ pro vstup $w \in In$,
- na definici velikosti vstupu (tj. definici funkce $size$).

Časová složitost algoritmu

Někdy se také určuje časová složitost algoritmu v **průměrném případě**:

- Musí se předpokládat určité **pravděpodobností rozdělení** na dané množině vstupů.
- Místo maximální doby výpočtu nad vstupy velikosti n se uvažuje střední hodnota doby těchto výpočtů.
- Většinou je analýza v průměrném případě o dost komplikovanější než analýza nejhoršího případu.
- Často se tyto dvě funkce příliš neliší, někdy je ale rozdíl významný.

Poznámka: Zkoumat složitost v **nejlepším případě** většinou moc smysl nemá.

Časová složitost algoritmu

Příklad analýzy časové složitosti algoritmu FIND-MAX **bez** použití asymptotické notace:

- Takto podrobně se analýza výpočetní složitosti algoritmu téměř nikdy **nedělá** — je to příliš pracné a komplikované.
- Uvidíme tak ale, co vše je při použití asymptotické notace zanedbáno a o kolik je analýza s použitím asymptotické notace jednodušší.
- Budeme počítat s konstantami $c_0, c_1, \ldots, c_8$, které udávají dobu trvání jednotlivých instrukcí — nebudeme počítat s konkrétními čísly.

Časová složitost algoritmu

Předpokládáme vstupy tvaru (A, n), kde A je pole a n počet prvků tohoto pole (přičemž $n \geq 1$).

Jako velikost vstupu (A, n) zvolme n.

Uvažujme nyní o nějaké jednom vstupu $w = (A, n)$ velikosti n:

- Dobu výpočtu $t(w)$ nad vstupem w můžeme vyjádřit jako

$$t(w) = c_0 m_0 + c_1 m_1 + \cdots + c_8 m_8,$$

kde $m_0, m_1, \ldots, m_8$ jsou čísla udávající, kolikrát je daná instukce při výpočtu nad vstupem w provedena.

Časová složitost algoritmu

Instr.	doba	počet provedení	hodnota m_i
$k := 0$a	c_0	m_0	1
$i := 1$	c_1	m_1	1
$[i < n]$	c_2	m_2	$n-1$
$[i \geq n]$	c_3	m_3	1
$[A[i] \leq A[k]]$	c_4	m_4	$n-1-\ell$
$[A[i] > A[k]]$	c_5	m_5	ℓ
$k := i$	c_6	m_6	ℓ
$i := i+1$	c_7	m_7	$n-1$
$result := A[k]$	c_8	m_8	1

ℓ — počet průchodů cyklem, kdy platí $A[i] > A[k]$ (zjevně je $0 \leq \ell < n$)

Časová složitost algoritmu

Dosazením do

$$t(w) = c_0 m_0 + c_1 m_1 + \cdots + c_8 m_8,$$

dostaneme

$$t(w) = d_1 + d_2 \cdot (n-1) + d_3 \cdot (n-1-\ell) + d_4 \cdot \ell,$$

kde

$$d_1 = c_0 + c_1 + c_3 + c_8 \qquad d_3 = c_4$$
$$d_2 = c_2 + c_7 \qquad d_4 = c_5 + c_6$$

Po úpravě je

$$t(w) = (d_2 + d_3) \cdot n + (d_4 - d_3) \cdot \ell + (d_1 - d_2 - d_3)$$

Poznámka: $t(w)$ není časová složitost, ale doba výpočtu pro konkrétní vstup w

Časová složitost algoritmu

Například pokud budou doby provedení jednotlivých instrukcí následující:

Instr.	označení	doba
$k := 0$	c_0	4
$i := 1$	c_1	4
$[i < n]$	c_2	10
$[i \geq n]$	c_3	12
$[A[i] \leq A[k]]$	c_4	14
$[A[i] > A[k]]$	c_5	12
$k := i$	c_6	5
$i := i + 1$	c_7	6
$result := A[k]$	c_8	5

bude $d_1 = 25$, $d_2 = 16$, $d_3 = 14$ a $d_4 = 17$.

V takovém případě je $t(w) = 30n + 3\ell - 5$.

Pro konkrétní vstup $w = ([3, 8, 4, 5, 2], 5)$ je $n = 5$ a $\ell = 1$, takže $t(w) = 30 \cdot 5 + 3 \cdot 1 - 5 = 148$.

Časová složitost algoritmu

Pro které vstupy velikosti n bude výpočet trvat nejdéle (tj. které vstupy představují nejhorší případ), může záviset na detailech implementace a přesných hodnotách konstant:

Doba výpočtu algoritmu FIND-MAX pro vstup $w = (A, n)$ velikosti n:

$$t(w) \;=\; (d_2 + d_3) \cdot n \;+\; (d_4 - d_3) \cdot \ell \;+\; (d_1 - d_2 - d_3)$$

- Pokud $d_3 \geq d_4$ — nejhorší jsou případy, kdy má ℓ co nejmenší hodnotu

  $\ell = 0$ — například vstupy tvaru $[0, 0, \ldots, 0]$ nebo třeba $[n,\ n-1,\ n-2,\ \ldots,\ 2,\ 1]$

- Pokud $d_3 \leq d_4$ — nejhorší jsou případy, kdy má ℓ co největší hodnotu

  $\ell = n - 1$ — například vstupy tvaru $[0, 1, \ldots, n-1]$

Časová složitost algoritmu

Časová složitost $T(n)$ algoritmu FIND-MAX v nejhorším případě je tedy dána následovně:

- Pokud $d_3 \geq d_4$:

$$T(n) = (d_2 + d_3) \cdot n + (d_1 - d_2 - d_3)$$

- Pokud $d_3 \leq d_4$:

$$\begin{aligned} T(n) &= (d_2 + d_3) \cdot n + (d_4 - d_3) \cdot (n-1) + (d_1 - d_2 - d_3) \\ &= (d_2 + d_4) \cdot n + (d_1 - d_2 - d_4) \end{aligned}$$

Příklad: Pro $d_1 = 25$, $d_2 = 16$, $d_3 = 14$ a $d_4 = 17$ bude

$$\begin{aligned} T(n) &= (16 + 17) \cdot n + (25 - 16 - 17) \\ &= 33n - 8 \end{aligned}$$

Časová složitost algoritmu

V obou případech (ať už $d_3 \geq d_4$ nebo $d_3 \leq d_4$) bude časová složitost algoritmu FIND-MAX funkce tvaru

$$T(n) = an + b$$

kde a a b jsou nějaké konstanty, jejichž přesné hodnoty závisí na délce trvání jednotlivých instrukcí.

Poznámka: Konkrétně bychom tyto konstanty mohli vyjádřit jako

$$a = d_2 + \max\{d_3, d_4\} \qquad b = d_1 - d_2 - \max\{d_3, d_4\}$$

Například

$$T(n) = 33n - 8$$

Časová složitost algoritmu

Pokud bychom se spokojili s tím, že časová složitost algoritmu FIND-MAX je nějaká funkce tvaru

$$T(n) = an + b,$$

kde by nás ale nezajímaly konkrétní hodnoty konstant a a b, celá analýza mohla být výrazně jednodušší.

- Ve skutečnosti ani většinou nechceme vědět, jak přesně funkce $T(n)$ vypadá (obecně to může být nějaká velmi komplikovaná funkce), a stačilo by nám, že víme, že hodnoty funkce $T(n)$ „zhruba“ odpovídají hodnotám nějaké funkce $S(n) = an + b$, kde a a b jsou nějaké konstanty.

Časová složitost algoritmu

U dané funkce $T(n)$ vyjadřující časovou nebo paměťovou složitost se tak většinou spokojíme s jejím přibližným vyjádřením — **odhadem**, kde

- zanedbáme méně významné členy

  (např. ve funkci $T(n) = 15n^2 + 40n - 5$ zanedbáme členy $40n$ a -5 a místo původní funkce budeme uvažovat jen o funkci $T(n) = 15n^2$),

- zanedbáme konstanty, kterými se násobí

  (např. místo funkce $T(n) = 15n^2$ budeme uvažovat o funkci $T(n) = n^2$)

- konstanty v exponentech ignorovat nebudeme — například je podstatný rozdíl mezi funkcemi $T_1(n) = n^2$ a $T_2(n) = n^3$.

- bude nás zajímat, jak se funkce $T(n)$ chová pro „velké“ hodnoty n, chování na malých hodnotách budeme ignorovat

Rychlost růstu funkcí

Program zpracovává vstup velikosti n.
Předpokládejme, že pro vstup velikosti n provede $T(n)$ operací, a že provedení jedné operace trvá $1\,\mu s$ ($10^{-6}\,s$).

	n							
$T(n)$	20	40	60	80	100	200	500	1000
n	20 µs	40 µs	60 µs	80 µs	0.1 ms	0.2 ms	0.5 ms	1 ms
$n \log n$	86 µs	0.213 ms	0.354 ms	0.506 ms	0.664 ms	1.528 ms	4.48 ms	9.96 ms
n^2	0.4 ms	1.6 ms	3.6 ms	6.4 ms	10 ms	40 ms	0.25 s	1 s
n^3	8 ms	64 ms	0.216 s	0.512 s	1 s	8 s	125 s	16.7 min.
n^4	0.16 s	2.56 s	12.96 s	42 s	100 s	26.6 min.	17.36 hod.	11.57 dní
2^n	1.05 s	12.75 dní	36560 let	$38.3 \cdot 10^9$ let	$40.1 \cdot 10^{15}$ let	$50 \cdot 10^{45}$ let	$10.4 \cdot 10^{136}$ let	–
$n!$	77147 let	$2.59 \cdot 10^{34}$ let	$2.64 \cdot 10^{68}$ let	$2.27 \cdot 10^{105}$ let	$2.96 \cdot 10^{144}$ let	–	–	–

Rychlost růstu funkcí

Uvažujme 3 algoritmy se složitostmi $T_1(n) = n$, $T_2(n) = n^3$, $T_3(n) = 2^n$. Náš počítač zvládne v reálném čase (kolik jsme ochotni počkat) 10^{12} kroků.

Složitost	Velikost vstupu
$T_1(n) = n$	10^{12}
$T_2(n) = n^3$	10^4
$T_3(n) = 2^n$	40

Rychlost růstu funkcí

Uvažujme 3 algoritmy se složitostmi $T_1(n) = n$, $T_2(n) = n^3$, $T_3(n) = 2^n$. Náš počítač zvládne v reálném čase (kolik jsme ochotni počkat) 10^{12} kroků.

Složitost	Velikost vstupu
$T_1(n) = n$	10^{12}
$T_2(n) = n^3$	10^4
$T_3(n) = 2^n$	40

Nyní počítač 1000 násobně zrychlíme. Zvládne tedy 10^{15} kroků.

Složitost	Velikost vstupu	Nárust
$T_1(n) = n$	10^{15}	$1000\times$
$T_2(n) = n^3$	10^5	$10\times$
$T_3(n) = 2^n$	50	$+10$

Asymptotická notace

V následujícím se zaměříme na funkce typu $f : \mathbb{N} \to \mathbb{R}$, kde:

- Hodnota $f(n)$ nemusí být definovaná pro všechny hodnoty $n \in \mathbb{N}$, ale musí existovat nějaká konstanta n_0 taková, že hodnota $f(n)$ je definovaná pro všechna $n \in \mathbb{N}$ taková, že $n \geq n_0$.

  Příklad: Funkce $f(n) = \log_2(n)$ není definovaná pro $n = 0$, ale pro všechna $n \geq 1$ už definovaná je.

- Musí existovat taková konstanta n_0, že pro všechny hodnoty $n \in \mathbb{N}$, kde $n \geq n_0$, platí $f(n) \geq 0$.

  Příklad: Pro funkci $f(n) = n^2 - 25$ platí $f(n) \geq 0$ pro všechna $n \geq 5$.

Asymptotická notace

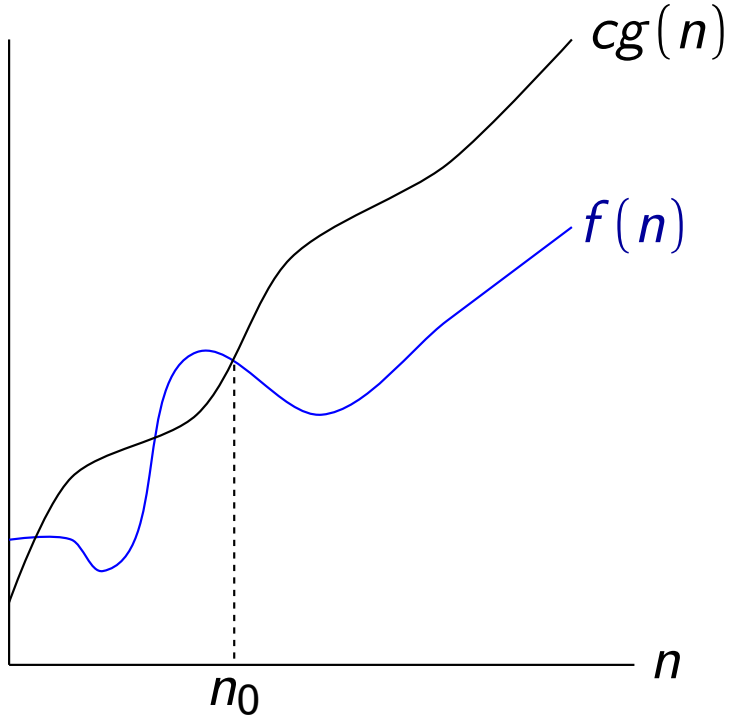
Vezměme si libovolnou funkci $f : \mathbb{N} \to \mathbb{R}$. Zápisy $O(f)$, $\Omega(f)$ a $\Theta(f)$ označují **množiny funkcí** typu $\mathbb{N} \to \mathbb{R}$, kde:

- $O(f)$ – množina všech funkcí, které rostou nejvýše tak rychle jako f
- $\Omega(f)$ – množina všech funkcí, které rostou alespoň tak rychle jako f
- $\Theta(f)$ – množina všech funkcí, které rostou stejně rychle jako f

Poznámka: Toto nejsou definice! Ty následují na následujících slidech.

- O – velké „O“
- Ω – velké řecké písmeno „omega“
- Θ – velké řecké písmeno „theta“

Asymptotická notace – symbol O



Definice

Vezměme si libovolnou funkci $g : \mathbb{N} \to \mathbb{R}$. Pro funkci $f : \mathbb{N} \to \mathbb{R}$ platí $f \in O(g)$ právě tehdy, když

$$(\exists c > 0)(\exists n_0 \geq 0)(\forall n \geq n_0)\big(f(n) \leq c\, g(n)\big).$$

Asymptotická notace – symbol O

Poznámky:

- c je kladné reálné číslo (tj. $c \in \mathbb{R}$ a $c > 0$)
- n_0 a n jsou přirozená čísla (tj. $n_0 \in \mathbb{N}$ a $n \in \mathbb{N}$)

Asymptotická notace – symbol O

Příklad: Vezměme si funkce $f(n) = 2n^2 + 3n + 7$ a $g(n) = n^2$.

Chceme ukázat $f \in O(g)$, tj. $f \in O(n^2)$:

- Postup 1:

  Zvolme například $c = 3$.

  $$cg(n) = 3n^2 = 2n^2 + \frac{1}{2}n^2 + \frac{1}{2}n^2$$

  Potřebujeme najít takové n_0, aby pro každé $n \geq n_0$ platilo současně

  $$2n^2 \geq 2n^2 \qquad \frac{1}{2}n^2 \geq 3n \qquad \frac{1}{2}n^2 \geq 7$$

  Snadno ověříme, že například $n_0 = 6$ vyhovuje těmto požadavkům.

  Pak pro každé $n \geq 6$ platí $cg(n) \geq f(n)$:

  $$cg(n) = 3n^2 = 2n^2 + \frac{1}{2}n^2 + \frac{1}{2}n^2 \geq 2n^2 + 3n + 7 = f(n)$$

Asymptotická notace – symbol O

Příklad, kde $f(n) = 2n^2 + 3n + 7$ a $g(n) = n^2$:

- Postup 2:

  Zvolme $c = 12$.

  $$cg(n) = 12n^2 = 2n^2 + 3n^2 + 7n^2$$

  Potřebujeme najít takové n_0, aby pro každé $n \geq n_0$ platilo současně

  $$2n^2 \geq 2n^2 \qquad 3n^2 \geq 3n \qquad 7n^2 \geq 7$$

  Uvedené vztahy zjevně platí pro $n_0 = 1$, takže pro každé $n \geq 1$ platí $cg(n) \geq f(n)$:

  $$cg(n) = 12n^2 = 2n^2 + 3n^2 + 7n^2 \geq 2n^2 + 3n + 7 = f(n)$$

Asymptotická notace – symbol O

Tvrzení

Předpokládejme, že a a b jsou nějaké konstanty takové, že $a > 0$ a $b > 0$, a k a ℓ jsou nějaké libovolné konstanty, kde $k \geq 0$, $\ell \geq 0$ a $k < \ell$.

Uvažujme funkce

$$f(n) = a \cdot n^k \qquad\qquad g(n) = b \cdot n^\ell$$

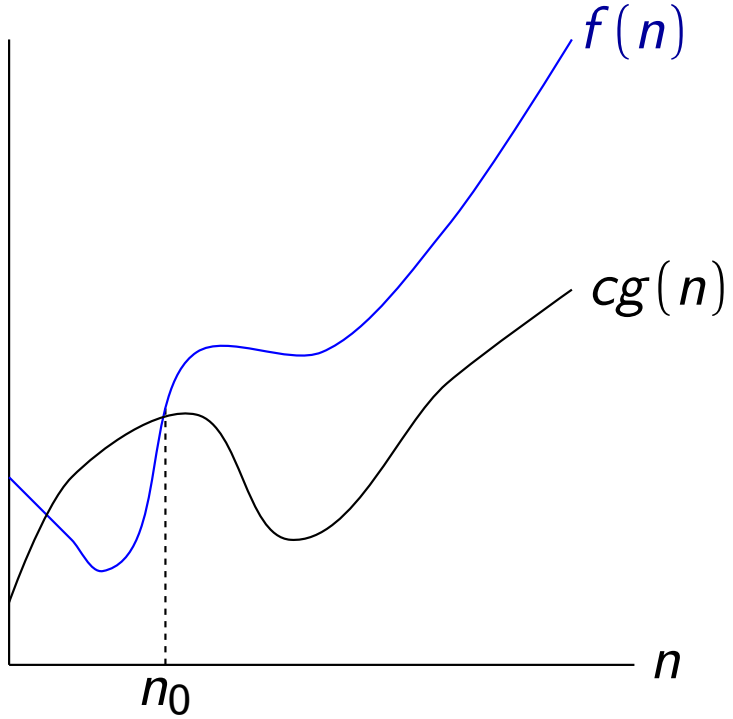
Pro každé takové funkce f a g platí $f \in O(g)$:

Důkaz: Zvolme $c = \frac{a}{b}$.

Vzhledem k tomu, že pro $n \geq 1$ zjevně platí $n^k \leq n^\ell$ (protože $k \leq l$), tak pro $n \geq 1$ platí

$$c \cdot g(n) = \frac{a}{b} \cdot g(n) = \frac{a}{b} \cdot b \cdot n^\ell = a \cdot n^\ell \geq a \cdot n^k = f(n)$$

Asymptotická notace – symbol Ω



Definice

Vezměme si libovolnou funkci $g : \mathbb{N} \to \mathbb{R}$. Pro funkci $f : \mathbb{N} \to \mathbb{R}$ platí $f \in \Omega(g)$ právě tehdy, když

$$(\exists c > 0)(\exists n_0 \geq 0)(\forall n \geq n_0)\big(c\, g(n) \leq f(n)\big).$$

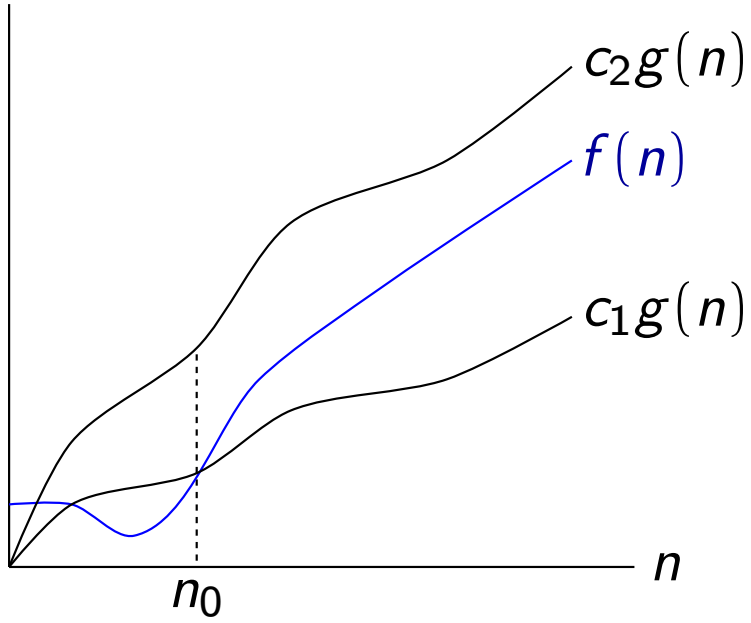
Asymptotická notace – symbol Ω

Není těžké zdůvodnit, že platí následující tvrzení:

Pro libovolné funkce f a g platí:

$$f \in O(g) \qquad \text{právě tehdy, když} \qquad g \in \Omega(f)$$

Asymptotická notace – symbol Θ



Definice

Vezměme si libovolnou funkci $g : \mathbb{N} \to \mathbb{R}$. Pro funkci $f : \mathbb{N} \to \mathbb{R}$ platí $f \in \Theta(g)$ právě tehdy, když

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

Asymptotická notace – symbol Θ

Pro libovolné funkce f a g platí:

$$f \in \Theta(g) \quad \text{právě tehdy, když} \quad f \in O(g) \text{ a } f \in \Omega(g)$$

$$f \in \Theta(g) \quad \text{právě tehdy, když} \quad f \in O(g) \text{ a } g \in O(f)$$

$$f \in \Theta(g) \quad \text{právě tehdy, když} \quad g \in \Theta(f)$$

Asymptotická notace

Pro libovolné tři funkce f, g a h platí:

- jestliže $f \in O(g)$ a $g \in O(h)$, pak $f \in O(h)$
- jestliže $f \in \Omega(g)$ a $g \in \Omega(h)$, pak $f \in \Omega(h)$
- jestliže $f \in \Theta(g)$ a $g \in \Theta(h)$, pak $f \in \Theta(h)$

Asymptotická notace

Příklady:

$n \in O(n^2)$
$1000n \in O(n)$
$2^{\log_2 n} \in \Theta(n)$
$n^3 \notin O(n^2)$
$n^2 \notin O(n)$
$n^3 + 2^n \notin O(n^2)$

$n^3 \in O(n^4)$
$0.00001n^2 - 10^{10}n \in \Theta(10^{10}n^2)$
$n^3 - n^2 \log_2^3 n + 1000n - 10^{100} \in \Theta(n^3)$
$n^3 + 1000n - 10^{100} \in O(n^3)$
$n^3 + n^2 \notin \Theta(n^2)$
$n! \notin O(2^n)$

Asymptotická notace

- Existují dvojice funkcí f a g takové, že

$$f \notin O(g) \qquad \text{a} \qquad g \notin O(f),$$

  například

$$f(n) = n \qquad\qquad g(n) = n^{1+\sin(n)}.$$

- $O(1)$ označuje množinu všech **omezených** funkcí, tj. funkcí jejichž funkční hodnoty jsou shora omezeny nějakou konstantou.

Asymptotická notace

- Pro libovolné dvě funkce f, g platí:
  - $\max(f, g) \in \Theta(f + g)$
  - pokud $f \in O(g)$, pak $f + g \in \Theta(g)$
- Pro libovolné čtyři funkce f_1, f_2, g_1, g_2 platí:
  - pokud $f_1 \in O(f_2)$ a $g_1 \in O(g_2)$, pak $f_1 + g_1 \in O(f_2 + g_2)$ a $f_1 \cdot g_1 \in O(f_2 \cdot g_2)$
  - pokud $f_1 \in \Theta(f_2)$ a $g_1 \in \Theta(g_2)$, pak $f_1 + g_1 \in \Theta(f_2 + g_2)$ a $f_1 \cdot g_1 \in \Theta(f_2 \cdot g_2)$

Asymptotická notace

- O funkci f řekneme, že je:

  logaritmická, pokud $f(n) \in \Theta(\log n)$
  lineární, pokud $f(n) \in \Theta(n)$
  kvadratická, pokud $f(n) \in \Theta(n^2)$
  kubická, pokud $f(n) \in \Theta(n^3)$
  polynomiální, pokud $f(n) \in O(n^k)$ pro nějaké $k > 0$
  exponenciální, pokud $f(n) \in O(c^{n^k})$ pro nějaké $c > 1$ a $k > 0$

- Exponenciální funkce se v asymptotické notaci často uvádí ve tvaru $2^{O(n^k)}$, protože potom již nemusíme uvažovat různé základy mocniny.

Asymptotická notace

Jak bylo uvedeno, výrazy $O(g)$, $\Omega(g)$ a $\Theta(g)$ označují určité množiny funkcí.

V odborných textech se však někdy používají tyto výrazy i v poněkud odlišném významu:

- zápis $O(g)$, $\Omega(g)$ nebo $\Theta(g)$ nereprezentuje danou množinu funkcí, ale **nějakou** funkci z dané množiny.

Tato konvence se používá zejména v zápisu rovnic nebo nerovnic.

Příklad: $3n^3 + 5n^2 - 11n + 2 = 3n^3 + O(n^2)$

Při použití této konvence je tedy možné například psát $f = O(g)$ místo $f \in O(g)$.

Složitost algoritmů

Řekněme, že bychom chtěli analyzovat časovou složitost $T(n)$ nějakého algoritmu, který se skládá z instrukcí $I_1, I_2, \ldots, I_k$:

- Pokud $m_1, m_2, \ldots, m_k$ jsou počty provedení jednotlivých instrukcí pro nějaký vstup w (tj. pro vstup x se instrukce I_i provede m_i krát), tak celkový počet instrukcí provedených pro vstup w je
$$T(n) = c_1 m_1 + c_2 m_2 + \cdots + c_k m_k.$$
- Vezměme si funkce $f_1, f_2, \ldots, f_k$, kde $f_i : \mathbb{N} \to \mathbb{R}$, přičemž $f_i(n)$ je maximum z počtu provedení instrukce I_i pro všechny vstupy velikosti n.
- Zjevně platí, že pro libovolnou funkci f_i je $T \in \Omega(f_i)$.
- Zjevně také platí $T \in O(f_1 + f_2 + \cdots + f_k)$.

Složitost algoritmů

- Připomeňme si, že pokud $f \in O(g)$, pak $f + g \in O(g)$.
- Pokud tedy pro některou funkci f_i platí, že pro všechny f_j, kde $j \neq i$, je $f_j \in O(f_i)$, pak

$$T \in O(f_i).$$

- Často se tedy při analýze celkové časové složitosti $T(n)$ můžeme omezit pouze na analýzu počtu provedení nejčastěji prováděné instrukce (pro vstup velikosti n je provedena maximálně $f_i(n)$ krát), protože platí

$$T \in \Theta(f_i).$$

Složitost algoritmů

Příklad: Při analýze složitosti algoritmu FIND-MAX jsme zjistili, že časová složitost daného algoritmu v nejhorším případě je

$$f(n) = an + b\,.$$

Kdybychom to nechtěli takto podrobně zjišťovat a spokojili se s hrubším odhadem, mohli jsme určit, že časová složitost tohoto algoritmu je $\Theta(n)$, protože:

- Algoritmus obsahuje jediný cyklus, který se pro vstup velikosti n provede vždy právě $(n-1)$ krát, tj. počet průchodů cyklem je v $\Theta(n)$.
- V rámci jednoho průchodu cyklem se provede několik instrukcí, jejichž počet je shora i zdola omezen nějakými konstantami nezávislými na velikosti vstupu.
- Ostatní instrukce se provedou jednou. K celkové složitosti tak přispívají přičtením nějaké konstanty.

Složitost algoritmů

Pokusme se analyzovat časovou složitost následujícího algoritmu:

Algoritmus 1: Třídění přímým vkládáním

```
1  INSERTION-SORT (A, n):
2  begin
3      for j := 1 to n − 1 do
4          x := A[j]
5          i := j − 1
6          while i ≥ 0 and A[i] > x do
7              A[i + 1] := A[i]
8              i := i − 1
9          end
10         A[i + 1] := x
11     end
12 end
```

Tj. chceme najít funkci $T(n)$ takovou, že časová složitost algoritmu INSERTION-SORT v nejhorším případě je v $\Theta(T(n))$.

Složitost algoritmů

Uvažujme vstupy velikosti n:

- Vnější cyklus **for** se provede $n-1$ krát.
- Vnitřní cyklus **while** se pro danou hodnotu j provede maximálně $(j-1)$ krát.
- Existují vstupy, pro které platí že pro každou hodnotu j od 2 do n se vnitřní cyklus **while** provede právě $(j-1)$ krát.
- V nejhorším případě se tedy cyklus **while** provede celkem m krát, kde

$$m = 1 + 2 + \cdots + (n-1) = (1 + (n-1)) \cdot \tfrac{n-1}{2} = \tfrac{1}{2}n^2 - \tfrac{1}{2}n$$

- Celková časová složitost algoritmu INSERTION-SORT v nejhorším případě je tedy $\Theta(n^2)$.

Složitost algoritmů

V předchozím případě jsme přesně spočítali celkový počet průchodů cyklem **while**.

Obecně to není vždy možné spočítat takto přesně nebo to může být hodně komplikované. Pokud nás zajímá jen asymptotický odhad, tak to často ani není nutné.

Složitost algoritmů

Pokud bychom například neuměli spočítat součet aritmetické posloupnosti, mohli bychom provést analýzu následovně:

- Vnější cyklus **for** se neprovede více než n krát, vnitřní cyklus **while** se při každé iteraci vnějšího cyklu provede maximálně n krát. Celkově se tedy vnitřní cyklus provede maximálně n^2 krát.

  Platí tedy $T \in O(n^2)$.

- Pro některé vstupy se při posledních $\lfloor n/2 \rfloor$ průchodech cyklem **for** provede cyklus **while** alespoň $\lceil n/2 \rceil$ krát.

  Pro některé vstupy se tedy cyklus **while** provede alespoň $\lfloor n/2 \rfloor \cdot \lceil n/2 \rceil$ krát.

  $\lfloor n/2 \rfloor \cdot \lceil n/2 \rceil \geq (n/2 - 1) \cdot (n/2) = \frac{1}{4}n^2 - \frac{1}{2}n$

  Platí tedy $T \in \Omega(n^2)$.