2. Časová složitost algoritmu

Algoritmizace

Jiří Balun

Obsah

Rekapitulace a motivace

Na minulé přednášce

- základní pojmy: problém, instrukce a algoritmus
- pseudokód: proměnné, větvení programu, cykly
- analýza problému největšího společného dělitele

Na této přednášce

- datová struktura pole
- časová složitost algoritmu: v nejhorším případě a v průměrném případě
- srovnání funkcí, které se často vyskytují jako složitosti algoritmů
- asymptotická složitost: $\mathcal{O}$-notace, Ω-notace, Θ-notace a ostré meze

Datová struktura pole

Proč potřebujeme datové struktury?

Intuice: algoritmy pracují s daty – například je potřebují je uložit, vyhledat v nich hodnotu, setřídit nebo jinak zpracovat. **Datová struktura** určuje, jakým způsobem jsou data uložena v paměti a jak efektivně s nimi můžeme pracovat.

Motivace

Proč jsou datové struktury důležité:

- volba vhodné datové struktury může zásadně ovlivnit rychlost programu,
- stejný problém lze vyřešit různými způsoby:
  - hledání prvku v neuspořádaném poli – je potřeba projít všechny prvky,
  - hledání v setříděném poli – nemusíme prohledávat prvky, které jsou větší než hledaná hodnota, ale existuje i lepší způsob – opakovaně půlit interval hledání,
  - hledání v tabulce s přímým přístupem (hashovací tabulka) – často stačí jediný krok,
- dobrá datová struktura je základem pro efektivní algoritmus.

- datové struktury umožňují uchovávat a organizovat informace různými způsoby: pole, seznamy, stromy, zásobníky, fronty, grafy aj.
- různé datové struktury jsou navrženy pro různé typy úloh

Datová struktura: pole

Intuice: pole je jedna z nejjednodušších a nejčastěji používaných datových struktur. Umožňuje ukládat konečnou posloupnost hodnot, které jsou přístupné podle svého **indexu**.

Definice

Pole je datová struktura, která reprezentuje uspořádanou n-tici prvků stejného typu:

$$A = \langle a_0, a_1, \ldots, a_{n-1} \rangle$$

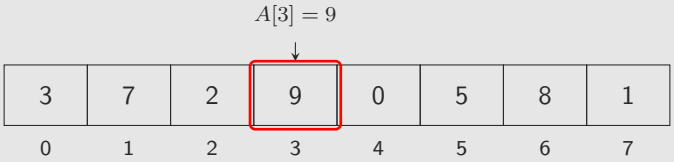
Každý prvek má svůj index i, který určuje jeho pozici (od 0 do $n-1$).

- indexování začíná obvykle od 0, to samé platí například i pro Python, C, Java,...
- délka pole $=$ počet prvků v poli, tj. číslo n
- hodnoty $a_0, a_1, \ldots, a_{n-1}$ uložené v poli lze **číst i změnit**
- **přímý** přístup k prvkům – prvek na pozici i získáme v konstantním čase (nezávisí na n)
  - pole obsahuje prvky stejného typu, proto lze uložit každou hodnotu do stejného počtu bitů
  - adresu v paměti prvku na indexu i vypočítáme: $AdresaPrvnihoPrvku + i * VelikostTypu$

Přístup k prvkům pole

Intuice: pole funguje podobně jako „přihrádky v řadě za sebou“ – každá přihrádka má své číslo, podle kterého do ní můžeme sáhnout.

Příklad



Mějme pole $A = \langle 3, 7, 2, 9, 0, 5, 8, 1 \rangle$, pro které

- $A[1]$ je zápis pro přístup k prvku na indexu 1 (má hodnotu 7),
- $A[3] \leftarrow 9$ je zápis pro změnu prvku na indexu 3 na hodnotu 9,
- $A[3] < A[5]$ je porovnání prvku na indexu 3 s prvkem na indexu 5 (tj. porovnání $9 < 5$).

Algoritmus: hledání minima v poli

Intuice: chceme najít nejmenší prvek v (nesetříděném) poli A o délce n. Projdeme postupně všechny prvky a zapamatujeme si zatím nejmenší nalezenou hodnotu.

Algorithm min-array

Input array $A = \langle a_0, \ldots, a_{n-1} \rangle$
Output minimum value from array A

```
1: min ← A[0]
2: for i ← 1 to n − 1 do
3:     if A[i] < min then
4:         min ← A[i]
5: return min
```

- pro jednoduchost předpokládáme, že pole obsahuje alespoň jeden prvek, jinak by instrukce na řádku 1 skončila chybou
- neznáme přesný počet instrukcí min-array pro pole délky n, algoritmus provede $n - 1$ porovnání (na řádku 3), ale nevíme kolikrát bude podmínka pravdivá

Příklad: průběh algoritmu min-array

Příklad

1: $min \leftarrow A[0]$
2: **for** $i \leftarrow 1$ **to** $n-1$ **do**
3: **if** $A[i] < min$ **then**
4: $min \leftarrow A[i]$
5: **return** min

Mějme pole $A = \langle 3, 7, 2, 9, 0, 5, 8, 1 \rangle$**, pro které** min-array(A) **vrátí hodnotu** 0**:**

i	$A[i]$	min (před)	Porovnání	min (po)
0	3	–	inicializace	3
1	7	3	$7 < 3$	3
2	2	3	$2 < 3$	2
3	9	2	$9 < 2$	2
4	0	2	$0 < 2$	0
5	5	0	$5 < 0$	0
6	8	0	$8 < 0$	0
7	1	0	$1 < 0$	0

Časová složitost algoritmu

Velikost instance

Intuice: chceme popsat, jak „velký“ je konkrétní vstup algoritmu, abychom mohli vyjádřit, jak se doba výpočtu mění s rostoucí velikostí vstupu.

Definice

Velikost instance (označujeme $|I|$) je přirozené číslo, které udává, jak rozsáhlý je vstup I daného problému. Formálně jde o zobrazení:

$$|\cdot| : \text{Vst} \to \mathbb{N},$$

kde Vst je množina všech přípustných vstupů problému.

- volba $|I|$ závisí na charakteru problému:
  - hledání minima pole: $|I| = n$ (počet prvků)
  - hledání nejkratší cesty v grafu: $|I| =$ počet uzlů nebo hran
  - výpočet sčítání dvou n-ciferných čísel: $|I| = n$
  - pro jedno číslo N se často bere: $|N| = \lfloor \log_2 N \rfloor + 1$ (počet bitů jeho zápisu)
- cílem je, aby doba běhu $t_{\mathcal{A}}(I)$ algoritmu $\mathcal{A}$ na vstupu I přirozeně závisela na $|I|$

Časová složitost algoritmu

Intuice: chceme změřit, kolik kroků algoritmus provede při zpracování vstupu velikosti n. Tím zjistíme, jak se doba výpočtu mění s velikostí instance $|I|$.

Definice

Časová složitost algoritmu $\mathcal{A}$ je funkce $T_{\mathcal{A}} : \mathbb{N} \to \mathbb{N}$:

$$T_{\mathcal{A}}(n) = \text{počet elementárních kroků, které } \mathcal{A} \text{ vykoná pro vstup velikosti } n.$$

- budeme spát jen $T(n)$ místo $T_{\mathcal{A}}(n)$, pokud bude algoritmus $\mathcal{A}$ jasný z kontextu
- funkce $T_{\mathcal{A}}(n)$ popisuje, jak algoritmus chová pro všechny vstupy o velikosti n
- obvykle nás zajímá **nejhorší případ** (maximum z všech vstupů) nebo **průměrný případ** (očekávaný počet kroků pro náhodný vstup)
- časová složitost udává závislost algoritmu na velikosti vstupu, ne skutečný čas běhu
- konkrétní běh $\mathcal{A}$ má délku $t_{\mathcal{A}}(I)$ – počet provedených instrukcí pro vstup I

Časová složitost v nejhorším případě

Intuice: chceme znát, jak dlouho může algoritmus trvat v nejhorší možné situaci – tedy pro takový vstup, na kterém pracuje nejdéle.

Definice

Časová složitost v nejhorším případě algoritmu A je funkce

$$T_{\mathcal{A}}(n) = \max\{\, t_{\mathcal{A}}(I) \mid I \text{ je vstup a } |I| = n \,\},$$

kde $t_{\mathcal{A}}(I)$ je počet elementárních kroků, které algoritmus $\mathcal{A}$ vykoná na vstupu I.

- udává **horní odhad** doby běhu algoritmu pro vstupy délky n
- říká nám, že algoritmus nikdy nebude trvat déle než $T_{\mathcal{A}}(n)$
- nejpoužívanější metrika, protože umožňuje bezpečné porovnávání algoritmů

Časová složitost v průměrném případě

Intuice: chceme odhadnout, jak dlouho algoritmus běží v průměru, nikoli pouze v nejhorší situaci. Zajímá nás jeho typické chování pro běžné vstupy.

Definice

Časová složitost v průměrném případě algoritmu $\mathcal{A}$ je očekávaný počet elementárních kroků pro všechny vstupy délky n:

$$T_{\mathcal{A}}(n) = \frac{t_{\mathcal{A}}(I_1) + t_{\mathcal{A}}(I_2) + \cdots + t_{\mathcal{A}}(I_m)}{m},$$

kde $I_1, \ldots, I_m$ jsou všechny možné vstupy délky n.

- udává **očekávaný čas běhu** algoritmu pro náhodně zvolený vstup délky n
- vyžaduje znalost nebo odhad pravděpodobnostního rozdělení vstupů
- poskytuje realističtější pohled na výkon algoritmu než nejhorší případ, ale může skrývat neschoponost algoritmu efektivně řešit některé vstupní instance

Jak je to s časovou složitostí min-array? (1/2)

Nejhorší případ

- hledáme vstupní instance $I_{\max}$, pro které trvá výpočet nejdéle
- v $I_{\max}$ je pole setřízeno sestupně, a proto je podmínka na řádku 3 platná v každé iteraci (v každé iteraci nalezneme nové minimum)

Analýza

- inicializace proměnné min a příkaz return: 2 kroky
- cyklus se provede $(n-1)$-krát a při každém opakování nastane:
  - 1 inkrementace i (v rámci cyklu for),
  - 1 porovnání $(A[i] < min)$
  - 1 vyhodnocení podmínky (vždy se vyhodnotí na pravdu)
  - 1 přiřazení $(min \leftarrow A[i])$
- celkový počet kroků v nejhorším případě:

$$T(n) = t(I_{\max}) = 2 + 4 \cdot (n-1) = 4n - 2$$

Jak je to s časovou složitostí min-array? (2/2)

Průměrný případ

- přiřazení $min \leftarrow A[i]$ se provádí jen tehdy, když je nalezen nový nejmenší prvek
- takových situací nebývá mnoho – daný prvek musí být menší než všechny předchozí, což je málo pravděpodobné, a proto s rostoucí velikostí pole přibývají velmi pomalu
- počet těchto přiřazení roste přibližně jako logaritmus z počtu prvků (přesná analýza průměrného případu je nad rámec tohoto kurzu)

Analýza

- počet kroků se liší od nejhoršího případu jen v počtu přiřazení na řádku 3
- pro vstupy $I_1, \ldots, I_m$ s velikostí n se průměrně provede $\log n$ přiřazení
- celkový počet kroků v průměrném případě:

$$T(n) \approx \frac{m(2 + 3 \cdot (n-1) + \log n)}{m} \approx 3n - 1 + \log n$$

Často se vyskytující složitosti

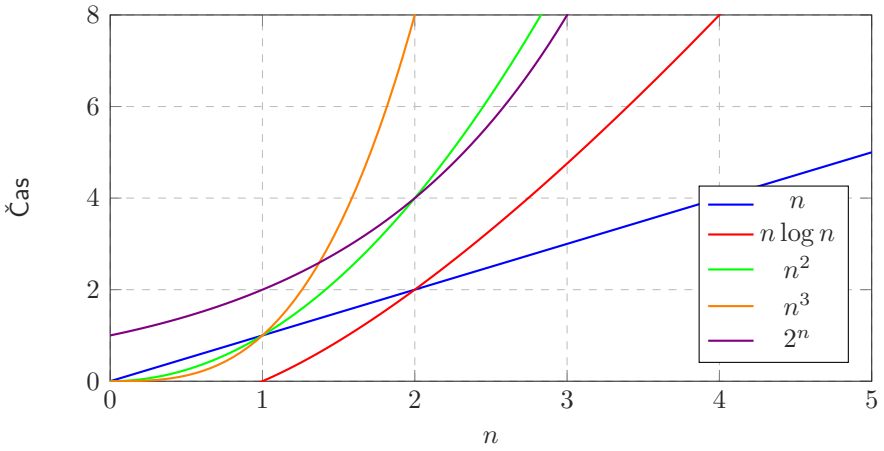
Intuice: složitosti různých algoritmů rostou s velikostí vstupu různě rychle. Některé zvládají i velké vstupy, jiné se stanou neproveditelnými už pro malé n.

Přehled běžných typů složitostí

Typ růstu	Příklad funkce	Typické algoritmy / operace
konstantní	c	přístup k prvku v poli
logaritmická	$\log n$	binární vyhledávání
lineární	n	prohledání pole, výpočet minima
lineárně–logaritmická	$n \log n$	efektivní třídění (MergeSort, QuickSort)
kvadratická	n^2	naivní třídění (BubbleSort, InsertionSort)
kubická	n^3	naivní násobení matic
exponenciální	2^n	úplné prohledávání stavového prostoru
faktoriální	$n!$	procházení všech permutací

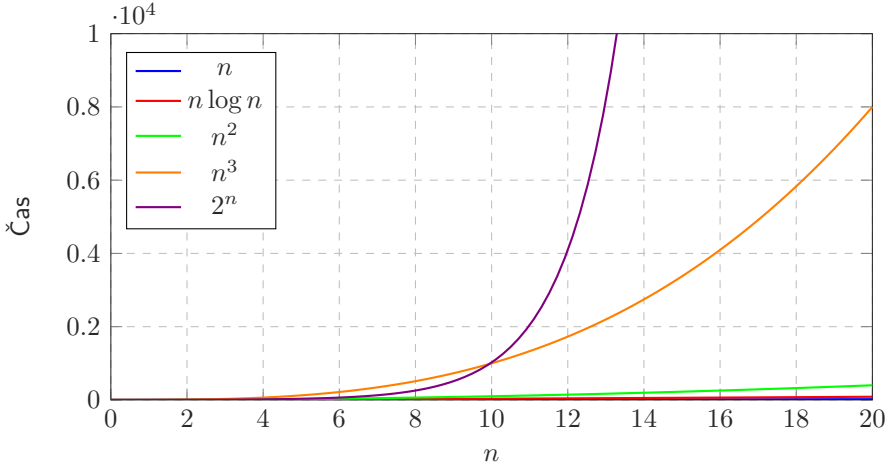
- rozdíly mezi těmito růsty se projeví velmi rychle už pro malá n

Srovnání základních funkcí (1/3)



Obrázek: grafické porovnání rychlosti růstu funkcí pro malé n.

Srovnání základních funkcí (2/3)



Obrázek: grafické porovnání rychlosti růstu funkcí pro větší n.

Srovnání základních funkcí (3/3)

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10	33	100	1000	1024	3628800
100	6.6	100	660	10^4	10^6	$1.27 \cdot 10^{30}$	$9.33 \cdot 10^{157}$
10^3	10	10^3	10^4	10^6	10^9	$1.07 \cdot 10^{301}$	$4.02 \cdot 10^{2567}$
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}	$1.99 \cdot 10^{3010}$	$2.85 \cdot 10^{35659}$
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2 \cdot 10^7$	10^{12}	10^{18}		

Tabulka 1: přibližné hodnoty základních funkcí.

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	0	0	0	0	0	0	0
100	0	0	0	0	0	$1.27 \cdot 10^{15}$	$9.33 \cdot 10^{142}$
10^3	0	0	0	0	0	$1.07 \cdot 10^{286}$	$4.02 \cdot 10^{2552}$
10^4	0	0	0	0	0.001	$1.99 \cdot 10^{2995}$	$2.85 \cdot 10^{35644}$
10^5	0	0	0	10^{-5}	1		
10^6	0	0	0	0.001	1000		

Tabulka 2: doba výpočtu v sekundách na velmi rychlém počítači (10^{15} operací za sekundu).

Asymptotická složitost

Motivace: proč hrubší analýza časové složitosti?

Intuice: přesné počítání instrukcí je zdlouhavé a pro velká n nepřináší podstatné informace. Důležitý je **řád růstu** funkce – jak rychle se zvětšuje s velikostí vstupu.

Příklad

Mějme algoritmus s časovou složitostí

$$T(n) = 2n^2 + 100n + 500$$

přičemž významnost jednotlivých členů:

- pro $n = 10$: $200 + \quad 1000 + 500 = 1700$
- pro $n = 100$: $20\,000 + \quad 10\,000 + 500 = 30\,500$
- pro $n = 1000$: $2\,000\,000 + 100\,000 + 500 = 2\,100\,500$

- pro malá n mohou mít nižší členy vliv, ale s rostoucím n převládá nejvyšší mocnina
- zajímá nás tedy jen hlavní člen, který určuje tempo růstu
- tento přístup vede k **asymptotické analýze** a zavedení $\mathcal{O}$-notace, Ω-notace a Θ-notace

$\mathcal{O}(g)$ – asymptotická horní mez

Intuice: pomocí $\mathcal{O}$-notace popisujeme, jak rychle roste časová složitost algoritmu. Funkce $f(n)$ patří do $\mathcal{O}(g(n))$, pokud od nějakého n dál je $f(n)$ menší než konstanta krát $g(n)$.

Definice

Pro funkci $g(n)$ je **asymptotická horní mez** (odhad) definována jako

$$\mathcal{O}(g(n)) = \{f(n) \mid \text{existuje } c > 0 \text{ a } n_0 \in \mathbb{N} \text{ takové, že pro každé } n \geq n_0 \text{ platí } 0 \leq f(n) \leq cg(n)\}$$

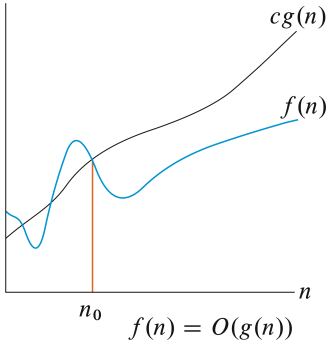
- píšeme běžně $f(n) = \mathcal{O}(g(n))$ místo přesnějšího $f \in \mathcal{O}(g(n))$
- používá se pro asymptotický odhad shora – určuje, jaký „řád růstu“ má algoritmus
- příklad: $2n^2 + 100n + 500 = \mathcal{O}(n^2)$

$\mathcal{O}(g)$ – asymptotická horní mez

Definice

Pro funkci $g(n)$ je **asymptotická horní mez** (odhad) definována jako

$$\mathcal{O}(g(n)) = \{f(n) \mid \text{existuje } c > 0 \text{ a } n_0 \in \mathbb{N} \text{ takové,}$$
$$\text{že pro každé } n \geq n_0 \text{ platí } 0 \leq f(n) \leq cg(n)\}$$



Zdroj obrázku: Introduction to Algorithms (Cormen et al.)

Příklad: asymptotická horní mez

Příklad (část 1/2)

Chceme ověřit, že funkce $f(n) = 2n^2 + 3n + 1$ patří do $\mathcal{O}(n^2)$, tedy že neroste rychleji než cn^2, kde c je nějaká vhodná konstanta.

Postup:

- musíme najít konstanty $c > 0$ a n_0, pro které platí

$$0 \leq f(n) \leq c\,n^2 \quad \text{pro všechna } n \geq n_0$$

- po dosazení $f(n)$ dostaneme nerovnici

$$2n^2 + 3n + 1 \leq c\,n^2$$

- kterou upravíme na tvar

$$(c-2)n^2 \geq 3n + 1$$

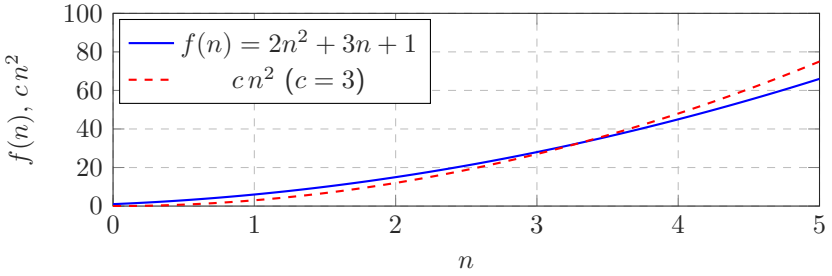
- hledáme takové c a n_0, pro které tato nerovnost platí pro všechna $n \geq n_0$.

Příklad (část 2/2)

- Z nerovnice $(c-2)n^2 \geq 3n+1$ plyne, že pro velká n se člen n^2 stává dominantním,
- nejprve zvolíme dostatečně velké c, aby pravá strana nerostla rychleji než levá,
- zvolme například $c = 3$, potom dostaneme

$$(3-2)n^2 = n^2 \geq 3n+1$$

- tato nerovnost platí pro všechna $n \geq 4$, takže můžeme zvolit $n_0 = 4$,
- tedy platí $f(n) \leq 3n^2$ pro všechna $n \geq 4$, a proto $f(n) = \mathcal{O}(n^2)$.



Příklad: kdy nerovnost neplatí?

Příklad

Uvažujme funkce $f(n) = n^2$ a $g(n) = n$. Chceme ověřit, zda $f(n) = \mathcal{O}(g(n))$.

Postup:

- podle definice musí existovat konstanty $c > 0$ a n_0, pro které platí

$$0 \leq f(n) \leq c\, g(n) \quad \text{pro všechna } n > n_0$$

- po dosazení:

$$n^2 \leq c\, n$$

  po vydělení n (pro $n > 0$) dostáváme

$$n \leq c$$

- tato nerovnost však **nemůže platit pro všechna** n, protože levá strana roste neomezeně, zatímco c je jenom konstanta,
- **nelze tedy najít takové** c **a** n_0, pro které by nerovnost platila, a proto $f(n) \neq \mathcal{O}(n)$.

$\Omega(g)$ – asymptotická dolní mez

Intuice: pomocí Ω-notace popisujeme, jak pomalu může algoritmus růst. Funkce $f(n)$ patří do $\Omega(g(n))$, pokud od nějakého n dál je $f(n)$ větší než konstanta krát $g(n)$.

Definice

Pro funkci $g(n)$ je **asymptotická dolní mez** (odhad zespodu) definována jako

$$\Omega(g(n)) = \{f(n) \mid \text{existuje } c > 0 \text{ a } n_0 \in \mathbb{N} \text{ takové, že pro každé } n \geq n_0 \text{ platí } 0 \leq cg(n) \leq f(n)\}$$

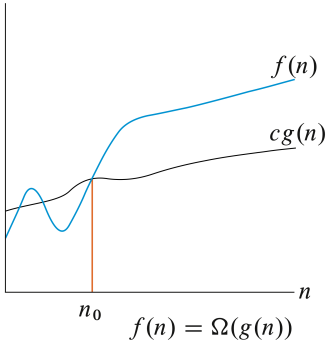
- běžně píšeme $f(n) = \Omega(g(n))$ místo přesnějšího $f \in \Omega(g(n))$
- asymptotický odhad zespodu – určuje, jak rychle algoritmus **alespoň** roste
- příklad: $2n^2 + 100n + 500 = \Omega(n^2)$

$\Omega(g)$ – asymptotická dolní mez

Definice

Pro funkci $g(n)$ je **asymptotická dolní mez** definována jako

$$\Omega(g(n)) = \{f(n) \mid \text{existuje } c > 0 \text{ a } n_0 \in \mathbb{N} \text{ takové, že pro každé } n \geq n_0 \text{ platí } 0 \leq cg(n) \leq f(n)\}$$



Zdroj obrázku: Introduction to Algorithms (Cormen et al.)

Příklad: asymptotická dolní mez

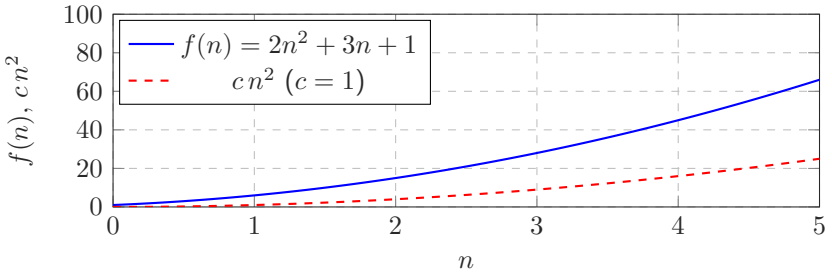
Příklad

Chceme ověřit, že funkce $f(n) = 2n^2 + 3n + 1$ patří do $\Omega(n^2)$, tedy že roste alespoň tak rychle jako cn^2, kde c je nějaká vhodná konstanta.

Řešení: existují konstanty $c > 0$ a n_0, pro které platí

$$0 \leq c\,n^2 \leq f(n) \quad \text{pro všechna } n \geq n_0.$$

Například $c = 1$ a $n_0 = 1$ splňují tuto podmínku. Pro všechna $n \geq 1$ je $n^2 \leq 2n^2 + 3n + 1$, tedy $f(n) = \Omega(n^2)$.



$\Theta(g)$ – asymptotická přesná mez

Intuice: pomocí Θ-notace popisujeme přesný řád růstu algoritmu. Funkce $f(n)$ patří do $\Theta(g(n))$, pokud roste stejně rychle (až na násobek konstanty) jako $g(n)$.

Definice

Pro funkci $g(n)$ je **asymptotická těsná mez** (oboustranný odhad) definována jako

$$\Theta(g(n)) = \{f(n) \mid \text{existují } c_1 > 0, c_2 > 0 \text{ a } n_0 \in \mathbb{N} \text{ takové, že pro každé } n \geq n_0 \text{ platí } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

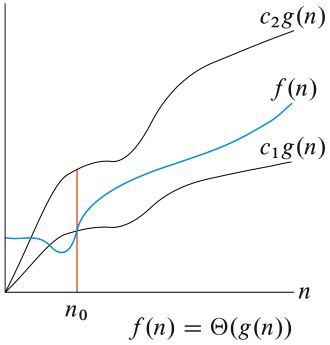
- běžně píšeme $f(n) = \Theta(g(n))$ místo přesnějšího $f \in \Theta(g(n))$
- $f(n) = \Theta(g(n))$ právě tehdy, když $f(n) = \mathcal{O}(g(n))$ a zároveň $f(n) = \Omega(g(n))$
- příklad: $2n^2 + 100n + 500 = \Theta(n^2)$

$\Theta(g)$ – asymptotická těsná mez

Definice

Pro funkci $g(n)$ je **asymptotická těsná mez** (oboustranný odhad) definována jako

$$\Theta(g(n)) = \{f(n) \mid \text{existují } c_1 > 0, c_2 > 0 \text{ a } n_0 \in \mathbb{N} \text{ takové, že pro každé } n \geq n_0 \text{ platí } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$



Zdroj obrázku: Introduction to Algorithms (Cormen et al.)

Příklad: asymptotická těsná mez

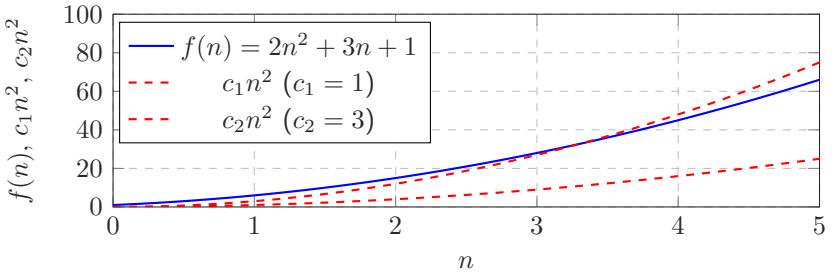
Příklad

Chceme ověřit, že funkce $f(n) = 2n^2 + 3n + 1$ patří do $\Theta(n^2)$, tedy že roste stejně rychle jako cn^2, kde c je nějaká vhodná konstanta.

Řešení: existují konstanty $c_1, c_2 > 0$ a n_0, pro které platí

$$0 \leq c_1 n^2 \leq f(n) \leq c_2 n^2 \quad \text{pro všechna } n \geq n_0.$$

Například $c_1 = 1$ a $c_2 = 3$ a $n_0 = 4$ splňují tuto podmínku. Pro všechna $n \geq 4$ je $n^2 \leq f(n) \leq 3n^2$, tedy $f(n) = \Theta(n^2)$.



Ostré (netěsné) meze – $o(g(n))$ a $\omega(g(n))$

Intuice: Notace $o(g(n))$ a $\omega(g(n))$ vyjadřují těsné (ostré) porovnání růstu funkcí. Na rozdíl od $\mathcal{O}$ (a Ω) zde $f(n)$ roste podstatně pomaleji (nebo podstatně rychleji) než $g(n)$.

Malé o – ostrá horní mez

Pro funkci $g(n)$ je $o(g(n))$ **asymptotická ostrá horní mez** a $\omega(g(n))$ je **asymptotická ostrá dolní mez**. Jsou definovány takto:

$$o(g(n)) = \{f(n) \mid \text{pro každé } c > 0 \text{ existuje } n_0 \in \mathbb{N} \text{ takové, že pro každé } n \geq n_0 \text{ platí } 0 \leq f(n) < cg(n)\}$$

$$\omega(g(n)) = \{f(n) \mid \text{pro každé } c > 0 \text{ existuje } n_0 \in \mathbb{N} \text{ takové, že pro každé } n \geq n_0 \text{ platí } 0 \leq cg(n) < f(n)\}$$

- $\mathcal{O}$, Ω, Θ – *neostré (včetně rovnosti)*, o, ω – *ostré (těsné)* meze (bez rovnosti)
- platí vztahy: $o(g) \subset \mathcal{O}(g)$ a $\omega(g) \subset \Omega(g)$
- například platí $3n = o(n^2)$, ale $3n^2 \neq o(n^2)$, protože cn^2 neroste rychleji než $3n^2$ pro všechny konstanty $c > 0$ (například pro $c = 4$)

Závěrem

Algoritmus min-array:

- min-array má lineární časovou složitost $\Theta(n)$
  - platí $T(n) = \mathcal{O}(n)$ – doba běhu roste nanejvýš lineárně
  - platí $T(n) = \Omega(n)$ – algoritmus musí projít všechny prvky pole
  - tedy zároveň $T(n) = \Theta(n)$
- pro libovolný vstup délky n provede přesně $n - 1$ porovnání
- složitost je stejná v nejlepším, průměrném i nejhorším případě

Doplňující materiály:

- na 2. slidech od prof. Bělohlávka najdete (od str. 28):
  - více příkladů na asymptotické meze
  - příklady na ostré meze
  - vztahy mezi jednotlivými pojmy