VŠB TECHNICKÁ UNIVERZITA OSTRAVA | FAKULTA ELEKTROTECHNIKY A INFORMATIKY | KATEDRA INFORMATIKY

Analýza složitosti algoritmů

doc. Mgr. Jiří Dvorský, Ph.D.

Stav prezentace ke dni 15. září 2024

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – TU Ostrava

Osnova přednášky

Analýza složitosti algoritmů

- Základy analýzy složitosti algoritmů
- Nejhorší, nejlepší a průměrný případ
- Asymptotické notace složitosti
- Analýza nerekurzivních algoritmů
- Analýza rekurzivních algoritmů

Analýza složitosti algoritmů

Základy analýzy složitosti algoritmů

Co analyzovat?

- správnost
- časová složitost
- prostorová složitost
- optimalita

Možné přístupy

- empirický a
- teoretický

Časová a prostorová složitost algoritmu

- **Časová složitost** – jak dlouho algoritmus bude pracovat.
- **Prostorová složitost** – kolik paměti bude algoritmus potřebovat **navíc** než je samotné uložení dat.
- Dříve byly oba zdroje kritické.
- Díky pokroku ve výpočetní technice je paměti relativně dost.
- Budeme zkoumat časovou složitost – tady lze dosáhnout významného pokroku.
- Ukazuje se, že prostorovou složitost lze zkoumat stejným aparátem jako časovou

Měření velikosti vstupu

- Triviální pozorování – větší data algoritmus obvykle zpracovává déle.
- Zavedeme parametr n označující velikost vstupních dat, který představuje například:
  - hledání v seznamu, poli – délka pole
  - vyhodnocení polynomu $p(x) = a_n x^n + \cdots + a_1 x + a_0$ v bodě x – stupeň polynomu
  - násobení matic typu $n \times n$ – rozměr matice. Skutečný počet čísel na vstupu je ale n^2, což je ale pořád závislé na n
  - kontrola pravopisu – počet znaků nebo počet slov, podle toho s čím algoritmus pracuje

Měření velikosti vstupu (pokrač.)

- test prvočíselnosti – vstupem je vždy jedno číslo (?!) a, doba běhu závisí na velikosti čísla (srovnej test 2^3 a 2^{64}), velikostí vstupu bude počet bitů nutných k zápisu čísla

$$n = \lfloor \log_2 a \rfloor + 1 \tag{2}$$

- grafové úlohy – počet vrcholů a/nebo počet hran – zde už jsou dva parametry

Empirické měření složitosti

- Zadáme vhodná (?) vstupní data a změříme dobu běhu programu v obvyklých jednotkách času.
- Nevýhody:
  - Závislost na konkrétním HW, způsobu implementace, kompilátoru.
  - Chceme měřit složitost algoritmů – nemáme prostředky pro zachycení výše uvedených vlivů.
  - Vývoj HW – znamená to, že se algoritmy zrychlují? Ne, ty zůstávají stejné.
  - Počet operací provedených programem lze obtížné zjistit.
  - Chceme se obejít bez implementace – zkoumáme přece algoritmy.

Časová složitost algoritmu

Časovou složitost algoritmu budeme vyjadřovat (měřit) **počtem vykonaných základních operací** vzhledem k (jako funkci) **velikosti vstupu** n:

$$T(n) \approx c_{op} C(n),$$

kde

- n je velikost vstupu,
- $T(n)$ je doba běhu algoritmu,
- c_{op} je doba vykonání jedné základní operace a
- $C(n)$ je počet základních operací.

Základní operace

Typické operace pro daný algoritmus, které významně přispívají ke celkové „době běhu“ algoritmu.

Problém	Měřítko vstupu	Základní operace
Hledání prvku v seznamu	Počet prvků v seznamu	Porovnání prvků
Násobení matic	Rozměry matic	Aritmetické operace (násobení)
Test prvočíselnosti	Počet bitů čísla	Dělení čísel
Grafové úlohy	Počet vrcholů a / nebo hran	Zpracování vrcholu či průchod hranou

Řádový růst složitosti

Ke vztahu

$$T(n) \approx c_{op} C(n),$$

je potřeba ale přistupovat „s rezervou“, protože

1. $C(n)$ nebere v úvahu vliv jiných operací než základních a
2. c_{op} nelze spolehlivě zjistit.

Vztah chápeme jako **rozumný odhad doby běhu algoritmu**, mimo extrémně malých n.

Řádový růst složitosti (pokrač.)

Problém

Kolikrát rychleji poběží můj algoritmus na počítači, který je **10×** rychlejší než můj současný počítač?

Řešení

Samozřejmě **10×**, c_{op} je desetinové.

Řádový růst složitosti (pokrač.)

Problém

Kolikrát déle poběží můj algoritmus pro dvojnásobně velký vstup, když $C(n) = \frac{1}{2}n(n-1)$?

Řešení

Aproximujeme shora počet operací $C(n)$

$$C(n) = \frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n < \frac{1}{2}n^2$$

a odtud

$$\frac{T(2n)}{T(n)} \approx \frac{c_{op}C(2n)}{c_{op}C(n)} \approx \frac{\frac{1}{2}(2n)^2}{\frac{1}{2}n^2} = \frac{4n^2}{n^2} = 4$$

Řádový růst složitosti (pokrač.)

Podstatné

Kvadratický růst složitosti – mluvíme o **řádovém růstu**.

Zanedbáno

Lineární člen $\frac{1}{2}n$

Řádový růst složitosti (pokrač.)

Tabulka ukazuje, jak rychle, či pomalu, rostou hodnoty vybraných funkcí pro různá n.

	Hodnota funkce tj. počet operací $C(n)$						
n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	$3,3$	10^1	$3,3 \cdot 10^1$	10^2	10^3	10^3	$3,6 \cdot 10^6$
10^2	$6,6$	10^2	$6,6 \cdot 10^2$	10^4	10^6	$1,3 \cdot 10^{30}$	$9,3 \cdot 10^{157}$
10^3	10	10^3	$1,0 \cdot 10^4$	10^6	10^9	n/a	n/a
10^4	13	10^4	$1,3 \cdot 10^5$	10^8	10^{12}	n/a	n/a
10^5	17	10^5	$1,7 \cdot 10^6$	10^{10}	10^{15}	n/a	n/a
10^6	20	10^6	$2,0 \cdot 10^7$	10^{12}	10^{18}	n/a	n/a

Pro malé n je rozdíl mezi hodnotami funkcí vcelku nezajímavý, ale pro zvětšující se n může být rozdíl propastný.

Řádový růst složitosti (pokrač.)

Poznámky

- Základ logaritmu není podstatný: $\log_a n = \log_a b \cdot \log_b n$.
- Počítači s rychlostí 10^{12} (tisíc miliard) operací za sekundu trvá provedení $2^{100} \approx 1,3 \cdot 10^{30}$ operací cca 40 miliard let. Stáří Země je cca 4,4 miliard let.
- O provedení **100!** operací nebudeme ani uvažovat...

Algoritmy s exponenciální nebo faktoriálovou řádovou složitostí jsou použitelné jen pro velice malé velikosti vstupu!

Řádový růst složitosti (pokrač.)

Problém

Kolikrát déle poběží můj algoritmus pro dvojnásobně velký vstup, pro algoritmy s různým řádovým růstem?

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
$2n$	$+1$	$2\times$	$\approx 2\times$	$4\times$	$8\times$	$(\dots)^2$	n/a

protože

$$\log_2(2n) = \log_2 2 + \log_2 n = 1 + \log_2 n$$

$$2^{2n} = (2^n)^2$$

Analýza složitosti algoritmů

Nejhorší, nejlepší a průměrný případ

Nejhorší, nejlepší a průměrný případ

- Počet základních operací udáváme jako funkci s jedním parametrem n, velikostí vstupu.
- Některé algoritmy můžou mít i pro stejné n různé počty zákl. operací, například algoritmus lineárního vyhledávání.

Vstup : Pole $A[0 \dots n-1]$ a hledaný prvek x
Výstup: Index prvního výskytu prvku x v poli A, jinak -1

```
1 for i ← 0 to n – 1 do
2 |   if A[i] = x then
3 |   |   return i;
4 |   end
5 end
6 return -1;
```

Nejhorší, nejlepší a průměrný případ (pokrač.)

Významné počty základních operací:

- $C_{worst}(n)$ – nejhorší případ, nejvyšší počet operací
- $C_{best}(n)$ – nejlepší případ, nejnižší počet operací
- $C_{avg}(n)$ – průměrný případ, průměrný počet operací.

Nejhorší případ $C_{worst}(n)$

- Analyzujeme algoritmus a hledáme vstup velikosti n pro který nastane nejvyšší možný počet operací.
- Nejhorší případ poskytuje horní mez složitosti, všechny ostatní případy jsou buď stejné nebo lepší.
- Nízký počet operací v nejhorším případě – pozitivní zpráva.

Příklad

Lineární vyhledávání: prvek x v poli A není nebo je nalezen až na konci, tedy $C_{worst}(n) = n$.

Nejlepší případ $C_{best}(n)$

- Obecně hledáme vstup velikosti n, pro který algoritmus vykoná nejmenší počet operací.
- Většinově nejlepší případ není tak důležitý jako nejhorší případ.
- Vstupy „podobné“, „blízké“ nejlepšímu. Třídění téměř setříděných posloupností.
- Nejlepší případ s „děsivým“ počtem operací – obecně špatná zpráva a „konečná“ pro algoritmus. Ale pro šifrovací algoritmus je „děsivý“ počet operací kryptoanalýzy i nejlepším případě nezbytný.

Příklad

Lineární vyhledávání: prvek x je prvním prvkem v poli A, $C_{best}(n) = 1$.

Průměrný případ $C_{avg}(n)$

- Počet operací v průměrném, „typickém“, „náhodném“ případě (nejlepší a nejhorší případy jsou extrémy).
- Nejedná se o průměr nejlepšího a nejhoršího případu!
- Musíme brát v úvahu pravděpodobnosti jednotlivých možných vstupů velikosti n.
- Analýza průměrného případu je tudíž komplikovanější než předchozích dvou.
- Existují algoritmy, kde se nejhorší a průměrný počet operací značně liší, např. QuickSort.

Průměrný případ $C_{avg}(n)$ – lineární vyhledávání

Předpoklady

1. pravděpodobnost úspěšného vyhledání p, kde $0 \leq p \leq 1$
2. pravděpodobnost nalezení na všech pozicích v poli je shodná a je rovna $\frac{p}{n}$

Průměrný případ $C_{avg}(n)$ – lineární vyhledávání (pokrač.)

Úspěšné vyhledání

- nalezení na první pozici – jedno porovnání s pravděpodobností $\frac{p}{n}$,
- nalezení na druhé pozice – dvě porovnání s pravděpodobností $\frac{p}{n}$, a tak dále, tedy

$$1\frac{p}{n} + 2\frac{p}{n} + \cdots + i\frac{p}{n} + \cdots + n\frac{p}{n}$$

Průměrný případ $C_{avg}(n)$ – lineární vyhledávání (pokrač.)

Neúspěšné vyhledání

- pravděpodobnost neúspěchu je $1 - p$ a provedeme n porovnání, tj. $n(1 - p)$

Odtud

$$\begin{aligned} C_{avg}(n) &= \left(1\frac{p}{n} + 2\frac{p}{n} + \cdots + i\frac{p}{n} + \cdots + n\frac{p}{n}\right) + n(1 - p) \\ &= \frac{p}{n}(1 + 2 + \cdots + i + \cdots + n) + n(1 - p) \\ &= \frac{p}{n}\left[\frac{1}{2}n(n + 1)\right] + n(1 - p) \\ &= \frac{1}{2}p(n + 1) + n(1 - p) \end{aligned}$$

Průměrný případ $C_{avg}(n)$ – lineární vyhledávání (pokrač.)

Rozbor

- vždy úspěšné hledání, $p = 1$ a tedy $C_{avg}(n) = \frac{1}{2}(n + 1)$
- neúspěšné hledání, $p = 0$ a tedy $C_{avg}(n) = n$

Amortizovaná složitost

- Nezkoumáme jeden, izolovaný, běh algoritmu, ale zkoumáme „sadu“ běhů s různými vstupy stejné velikosti.
- Zajímá nás celkový počet operací za sadu.
- Počet operací pro jeden vstup ze sady může být sice vysoký, ale je vyvážen, „amortizován“ výrazně menším počtem operací pro další vstupy ze sady.
- Například jeden ze vstupů způsobí rozsáhlou změnu v datové struktuře a díky tomu zpracování dalších vstupů proběhne snadněji.
- V průmyslu je například nákup drahého stroje amortizován levnější výrobou výrobku.

Zdroje pro samostatné studium

- Kniha [2], kapitola 2.1, strany 42 – 51
- Kniha [3], kapitola 2.2, strany 25 – 34 (částečně)

Analýza složitosti algoritmů

Asymptotické notace složitosti

O-notace

Definice

Mějme funkce $t(n)$ a $g(n)$, kde $t(n), g(n) : \mathbb{N} \to \mathbb{N}$. Říkáme, že funkce $t(n)$ patří do $O(g(n))$, jestliže existuje kladná nenulová reálná konstanta c a přirozené číslo $n_0 \geq 0$ takové, že

$$t(n) \leq cg(n)$$

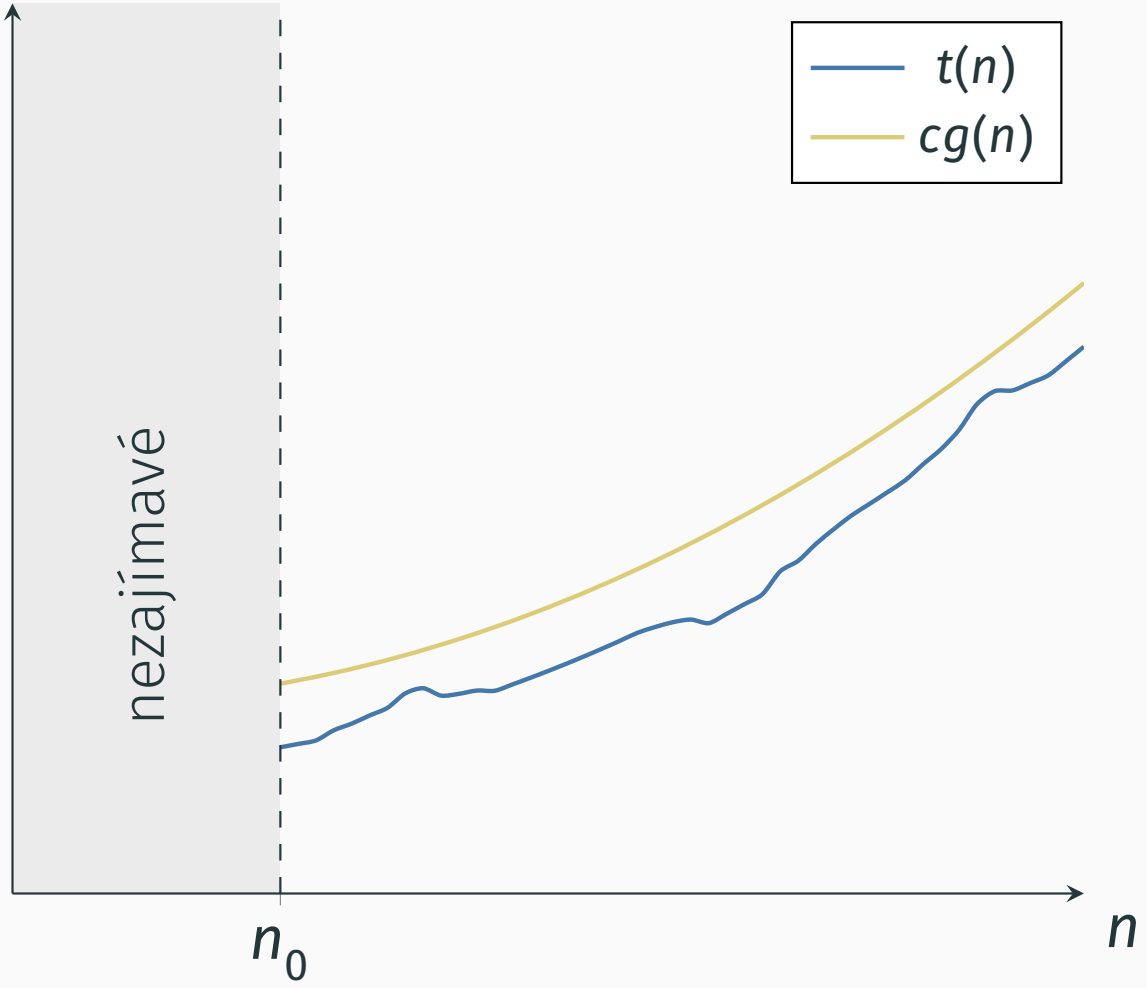
pro všechna $n \geq n_0$.

Poznámka

Místo „$t(n)$ patří do $O(g(n))$“ můžeme říkat, že „$t(n)$ je řádu $O(g(n))$“.

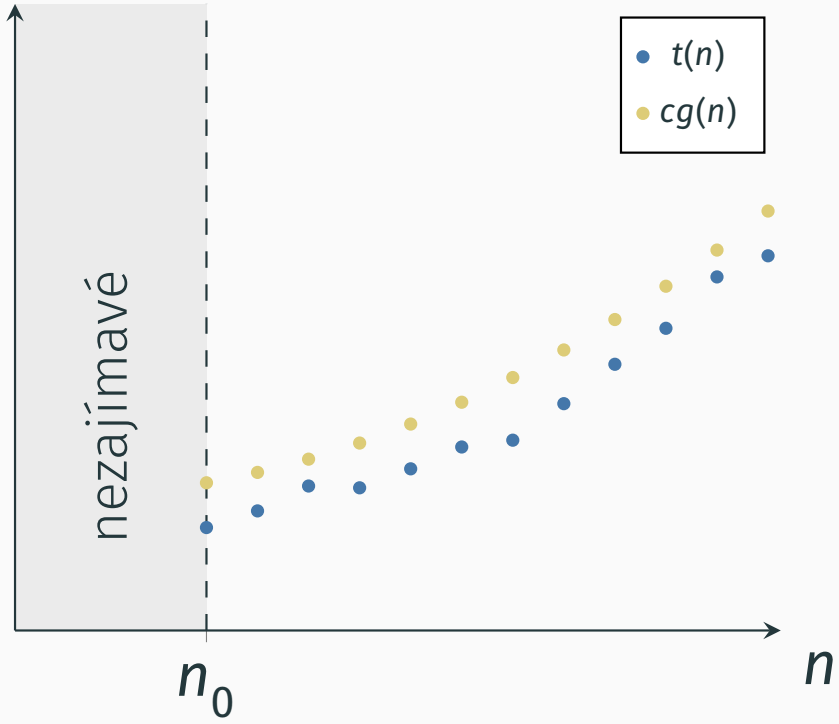
O-notace graficky

$$t(n) \in O(g(n))$$



O-notace – formálně správný graf

Formálně jsou definičním oborem i oborem hodnot funkcí $t(n)$ i $g(n)$ přirozená čísla ⇒ graf by měl být složen pouze z bodů, nikoliv křivek.



Proložíme-li body křivky ⇒ dostáváme spojité funkce ⇒ můžeme použít k výpočtům matematickou analýzu (limity, derivace atd.).

O-notace – příklad 1

Zadání

Dokažte, že $3n + 7 \in O(n)$.

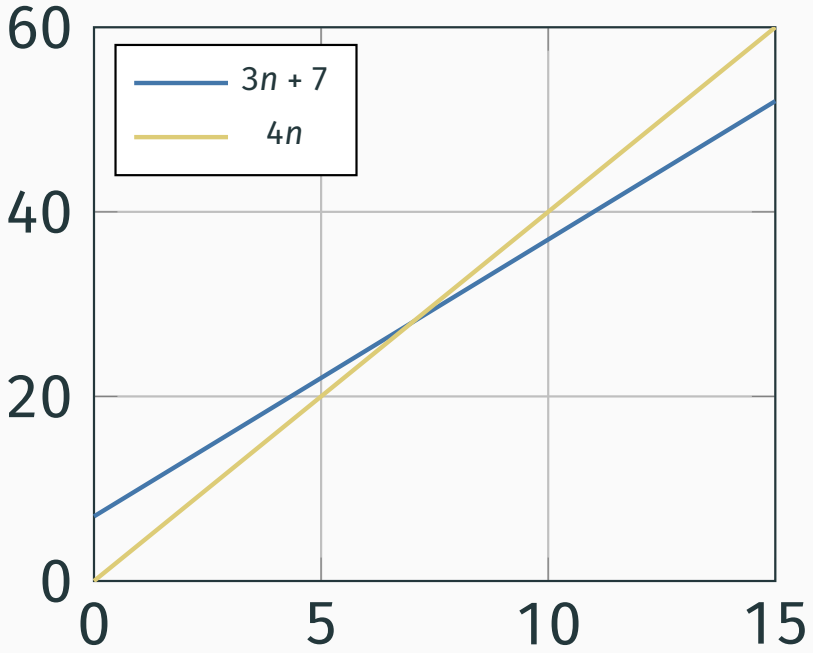
Řešení

1. Hledáme konstanty c a n_0 takové, aby platilo

$$3n + 7 \leq cn$$

pro všechna $n \geq n_0$.

2. Je zřejmé, že nutně $c > 3$. Zvolíme-li např. $c = 4$, pak $n_0 = 7$.



O-notace – příklad 2

Zadání

Dokažte, že

$3n + 7 \in O(n^2)$.

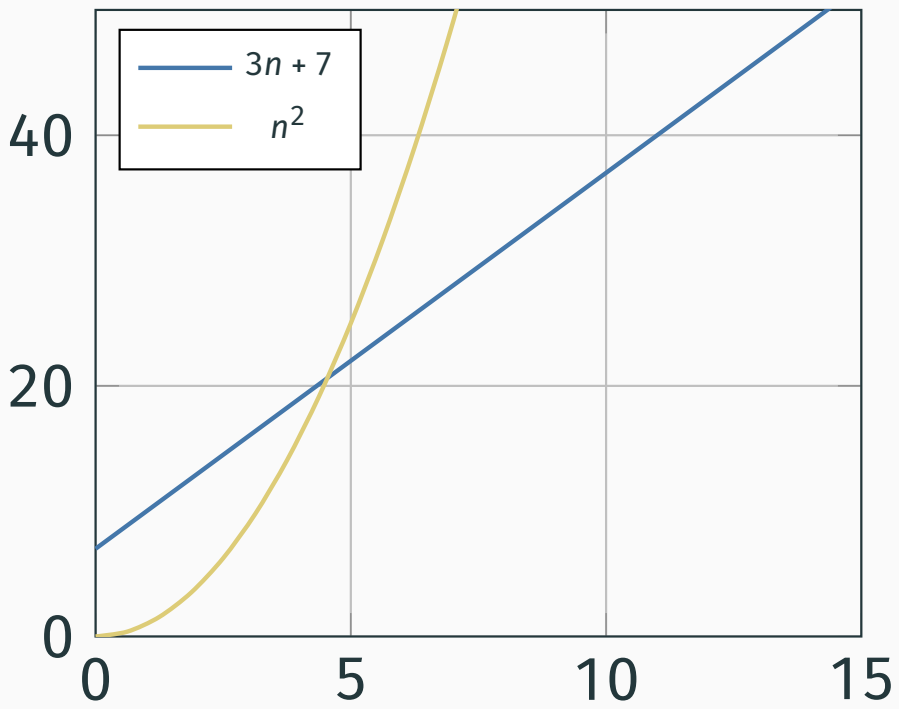
Řešení

1. Hledáme konstanty c a n_0 takové, aby platilo

$$3n + 7 \leq cn^2$$

pro všechna $n \geq n_0$.

2. Zvolíme-li $c = 1$, pak $n_0 = 5$.



O-notace – příklad 3

Zadání

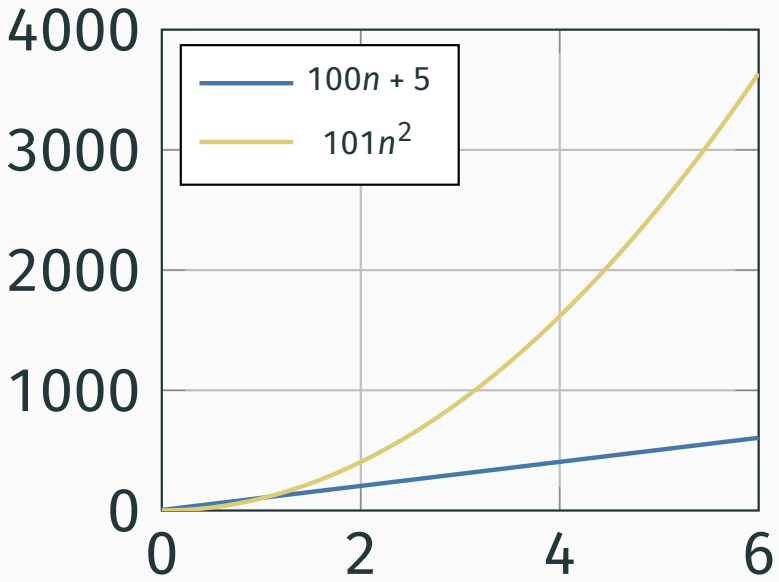
Dokažte, že $100n + 5 \in O(n^2)$.

Řešení

1. Platí, že $100n + 5 \leq 100n + n$ pro všechna $n \geq 5$.
2. Dále platí, že $101n \leq 101n^2$.
3. Odtud

$$100n + 5 \leq 101n \leq 101n^2$$

a tedy $c = 101$ a $n_0 = 5$.



O-notace – příklad 3 (pokrač.)

Důkaz lze vést i takto:

$$100n + 5 \leq 100n + 5n = 105n$$

pro všechna $n \geq 1$. Z toho plyne, že

$$105n \leq 105n^2$$

a tudíž $c = 105$ a $n_0 = 0$.

Definice O-notace neříká nic o jednoznačnosti hodnot c a n_0, pouze požaduje jejich existenci.

Ω-notace

Definice

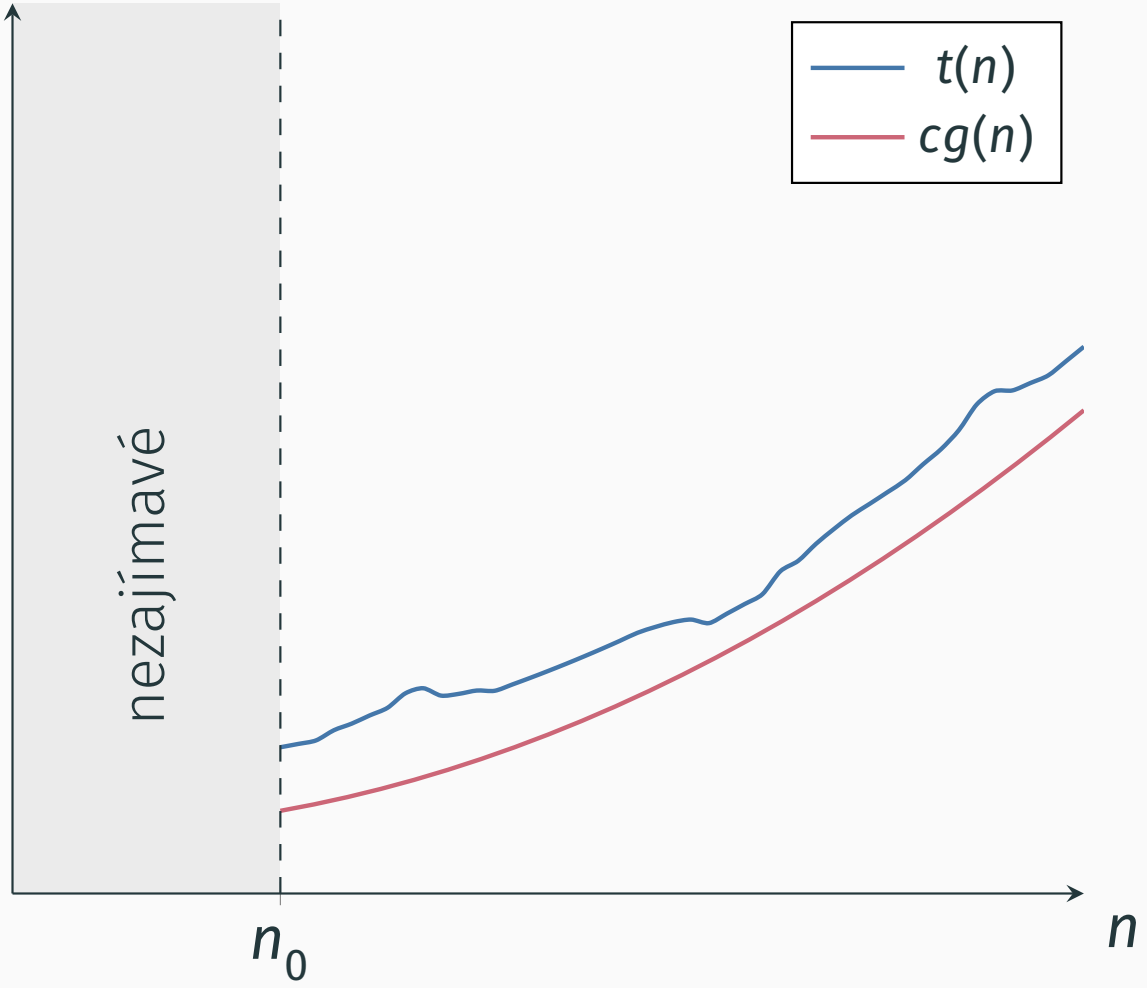
Mějme funkce $t(n)$ a $g(n)$, kde $t(n), g(n) : \mathbb{N} \to \mathbb{N}$. Říkáme, že funkce $t(n)$ patří do $\Omega(g(n))$, jestliže existuje kladná nenulová reálná konstanta c a přirozené číslo $n_0 \geq 0$ takové, že

$$t(n) \geq cg(n)$$

pro všechna $n \geq n_0$.

Ω-notace graficky

$$t(n) \in \Omega(g(n))$$



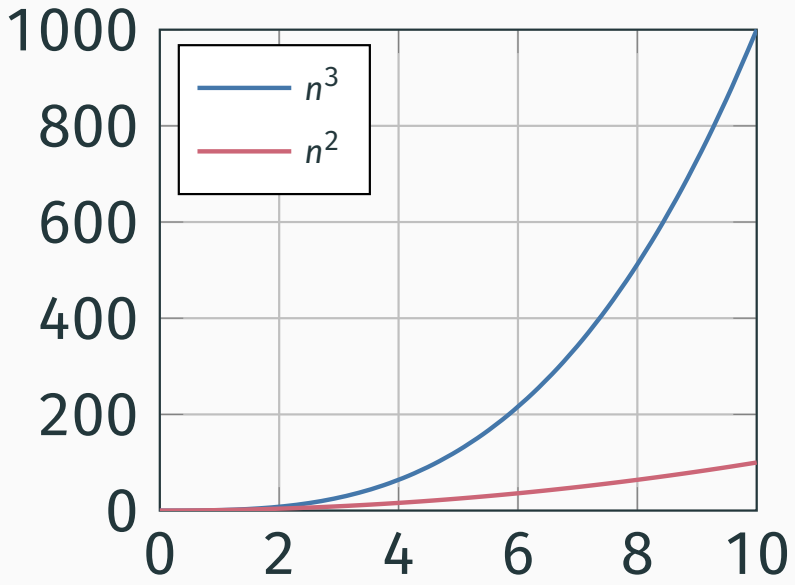
Ω-notace – příklad 1

Zadání

Dokažte, že platí $n^3 \in \Omega(n^2)$.

Řešení

1. Zřejmě platí, že $n^3 \geq n^2$ pro všechna $n \geq 0$.
2. Tudíž můžeme volit $c = 1$ a $n_0 = 0$.



Ω-notace – příklad 2

Zadání

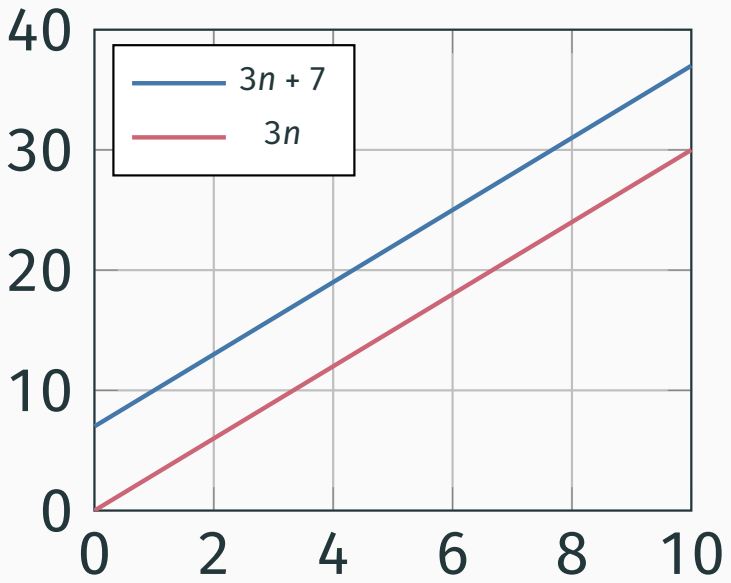
Dokažte, že platí $3n + 7 \in \Omega(n)$.

Řešení

1. Hledáme konstanty c a n_0 takové, aby platilo

   $$3n + 7 \geq cn$$

   pro všechna $n \geq n_0$.
2. Výraz $3n + 7 \geq 3n$ je platný pro všechna $n \geq 0$, tudíž $c = 3$ a $n_0 = 0$.



Θ-notace

Definice

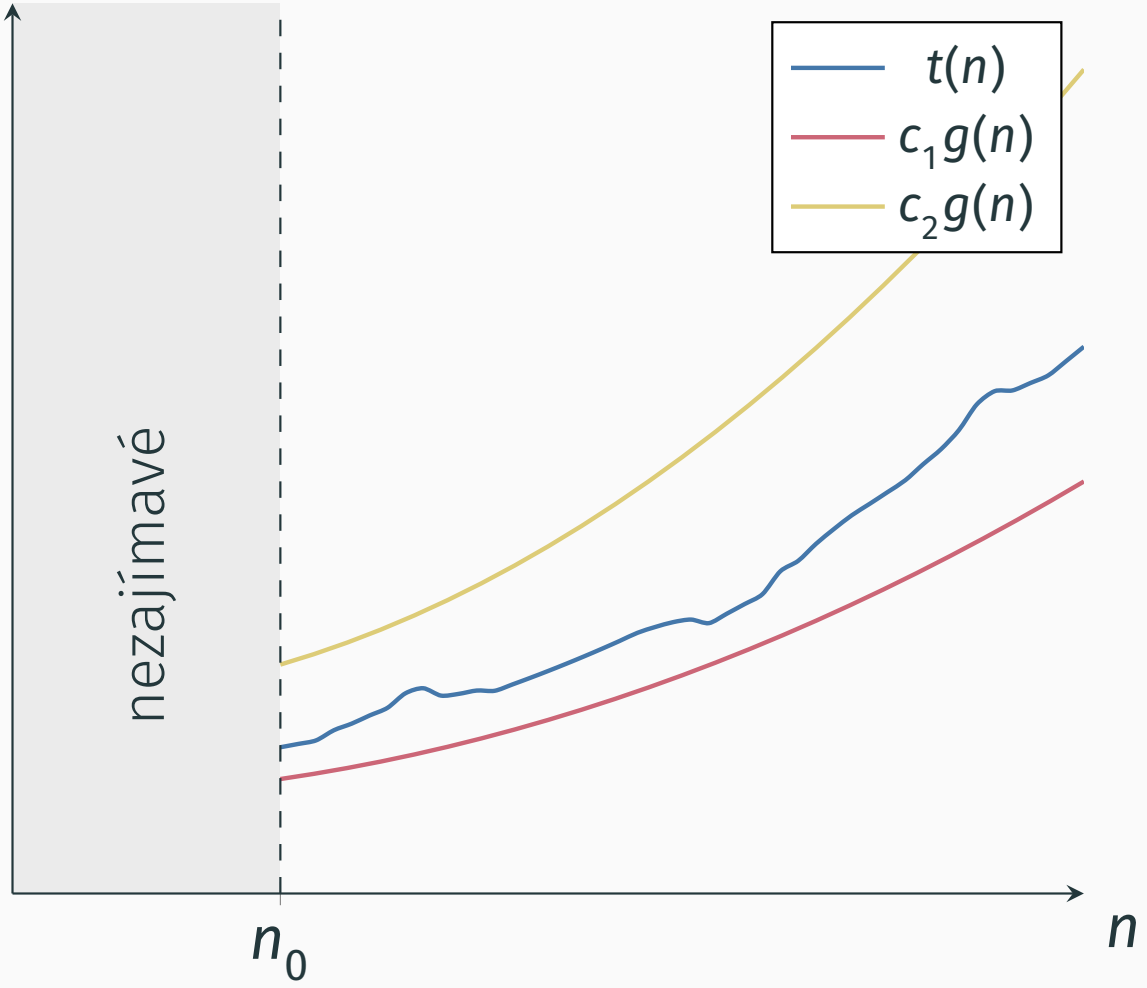
Mějme funkce $t(n)$ a $g(n)$, kde $t(n), g(n) : \mathbb{N} \to \mathbb{N}$. Říkáme, že funkce $t(n)$ patří do $\Theta(g(n))$, jestliže existují kladné nenulové reálné konstanty c_1, c_2 a přirozené číslo $n_0 \geq 0$ takové, že

$$c_1 g(n) \leq t(n) \leq c_2 g(n)$$

pro všechna $n \geq n_0$.

Θ-notace graficky

$$t(n) \in \Theta(g(n))$$



Θ-notace – příklad

Zadání

Dokažte, že $\frac{1}{2}n(n-1) \in \Theta(n^2)$.

Řešení

1. Nejprve dokážeme pravou nerovnost $t(n) \leq c_2 g(n)$ (omezení shora)

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \leq \frac{1}{2}n^2$$

   pro všechna $n \geq 0$.

Θ-notace – příklad (pokrač.)

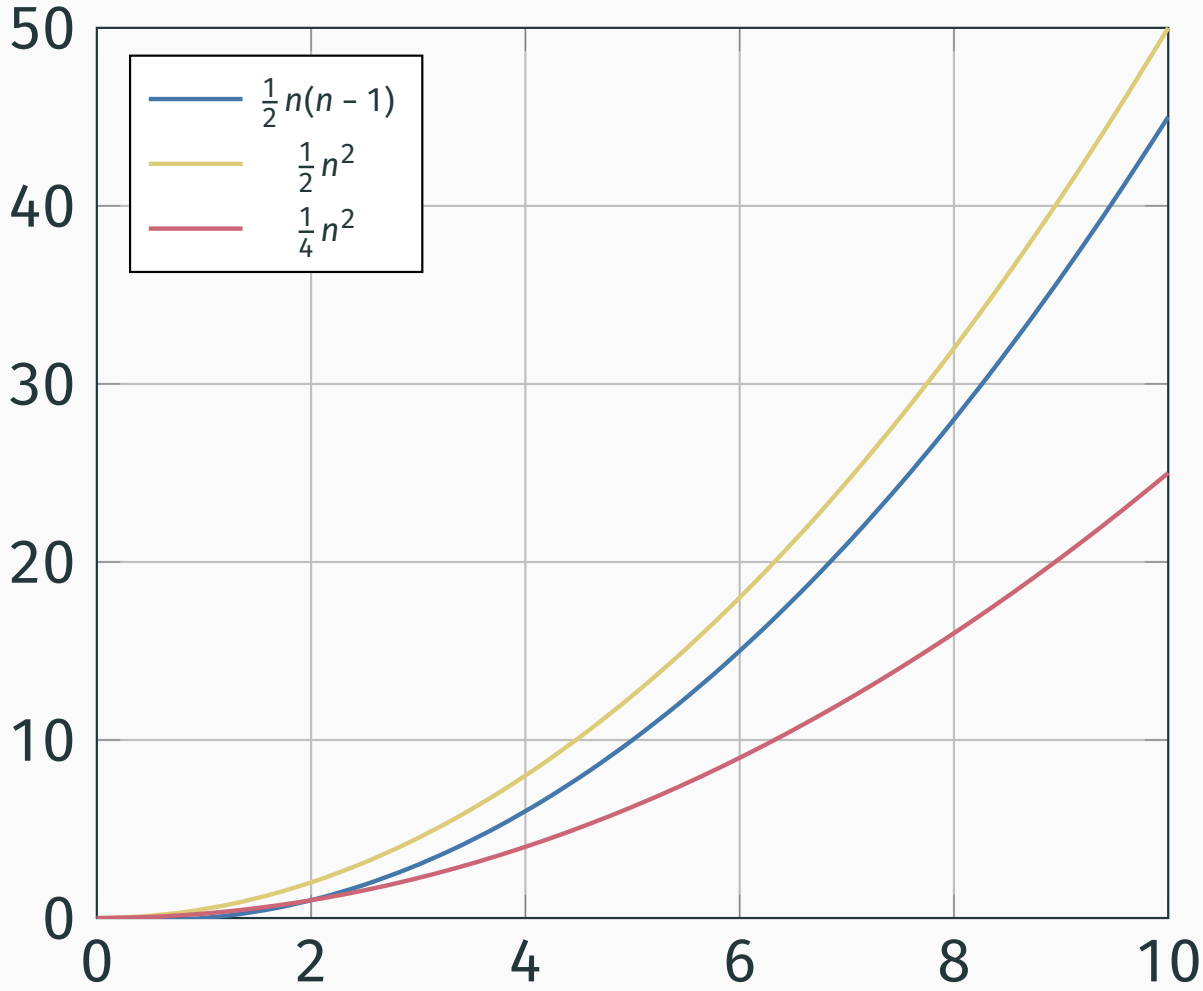
2. Levou nerovnost $c_1 g(n) \leq t(n)$ (omezení zdola) dokážeme takto:

$$\begin{aligned} t(n) = \frac{1}{2}n(n-1) &= \frac{1}{2}n^2 - \frac{1}{2}n \\ &\geq \frac{1}{2}n^2 - \frac{1}{2}n\frac{1}{2}n \\ &\geq \frac{1}{2}n^2 - \frac{1}{4}n^2 \\ &\geq \frac{1}{4}n^2 \end{aligned}$$

   Souhrnně tedy $\frac{1}{4}n^2 \leq \frac{1}{2}n(n-1)$ pro všechna $n \geq 2$.

3. Z předchozích nerovností plyne, že $c_1 = \frac{1}{4}$, $c_2 = \frac{1}{2}$ a $n_0 = 2$.

Θ-notace – příklad (pokrač.)



Vlastnosti asymptotické notace

Základní vlastnosti:

1. $f(n) \in O(f(n))$
2. $f(n) \in O(g(n)) \iff g(n) \in \Omega(f(n))$
3. $f(n) \in O(g(n)) \wedge g(n) \in O(h(n)) \implies f(n) \in O(h(n))$
4. $\Theta(f(n)) = O(f(n)) \wedge \Omega(f(n))$

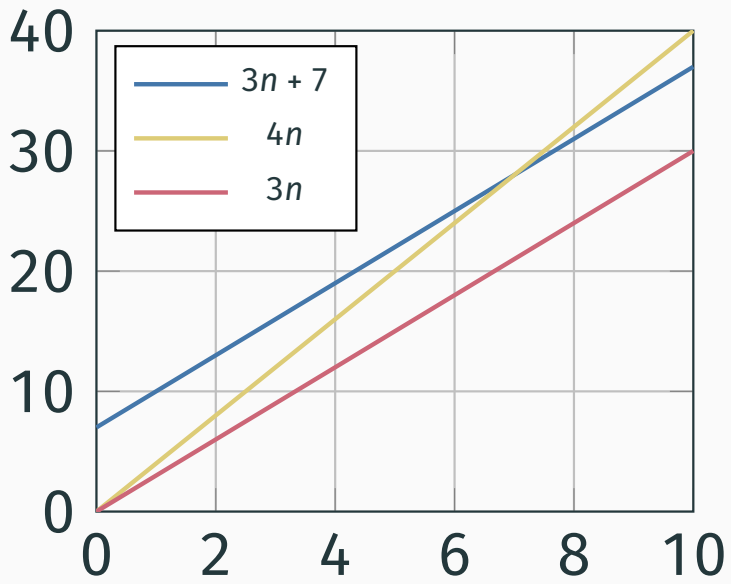
Vlastnosti asymptotické notace – využití

Zadání

Dokažte, že platí $3n + 7 \in \Theta(n)$.

Řešení

1. Z minulých příkladů víme, že $3n + 7 \in O(n)$ a současně $3n + 7 \in \Omega(n)$.
2. Proto platí, že $3n + 7 \in \Theta(n)$.
3. Konkrétně $c_1 = 3$, $c_2 = 4$ a $n_0 = 7$.



Vlastnosti asymptotické notace – výpočet složitosti

- Algoritmus A se skládá z částí A_1 a A_2.
- Části algoritmu se vykonávají po sobě, tj. po dokončení A_1 se začne vykonávat A_2.
- Složitost části A_1 je $t_1(n) \in O(g_1(n))$, složitost části A_2 je $t_2(n) \in O(g_2(n))$.
- Otázka zní – jaká je celková složitost algoritmu A?

Vlastnosti asymptotické notace – pomocné lemma

Lemma

Mějme libovolná reálná čísla a_1, a_2, b_1, b_2. *Potom platí*

$$a_1 \leq b_1 \wedge a_2 \leq b_2 \implies a_1 + a_2 \leq 2 \max(b_1, b_2).$$

Vlastnosti asymptotické notace – pomocné lemma (pokrač.)

Důkaz.

Z předpokladu víme, že

$$\begin{array}{rcl} a_1 & \leq & b_1 \\ a_2 & \leq & b_2 \\ \hline a_1 + a_2 & \leq & b_1 + b_2. \end{array}$$

Dále platí, že

$$b_1 + b_2 \leq 2 \max(b_1, b_2).$$

Odtud dostáváme

$$a_1 + a_2 \leq b_1 + b_2 \leq 2 \max(b_1, b_2).$$

□

Vlastnosti asymptotické notace – výpočet složitosti

Věta

Pokud $t_1(n) \in O(g_1(n))$ *a současně* $t_2(n) \in O(g_2(n))$*, potom*

$$t_1(n) + t_2(n) \in O(\max(g_1(n), g_2(n))).$$

Poznámka

Shodné tvrzení můžeme vyslovit i pro Ω a Θ notaci.

Vlastnosti asymptotické notace – výpočet složitosti (pokrač.)

Důkaz.

Protože $t_1(n) \in O(g_1(n))$, tak existuje kladná nenulová konstanta c_1 a nezáporná konstanta n_1 taková, že

$$t_1(n) \leq c_1 g_1(n) \qquad \forall n \geq n_1.$$

Obdobně

$$t_2(n) \leq c_2 g_2(n) \qquad \forall n \geq n_2.$$

Vlastnosti asymptotické notace – výpočet složitosti (pokrač.)

Důkaz.

Označme $c_3 = \max(c_1, c_2)$ a $n_0 \geq \max(n_1, n_2)$. Potom platí

$$\begin{aligned} t_1(n) + t_2(n) &\leq c_1 g_1(n) + c_2 g_2(n) \\ &\leq c_3 g_1(n) + c_3 g_2(n) = c_3[g_1(n) + g_2(n)] \\ &\leq 2c_3 \max(g_1(n), g_2(n)). \end{aligned}$$

Tudíž $t_1(n) + t_2(n) \in O(\max(g_1(n), g_2(n)))$, protože existují konstanty $c = 2c_3 = 2\max(c_1, c_2)$ a $n_0 = \max(n_1, n_2)$. □

Celkovou složitost algoritmu určuje část algoritmu s nejvyšší složitostí.

Vlastnosti asymptotické notace – výpočet složitosti, příklad

Zadání

Test, zda se v poli vyskytují dvě shodné hodnoty.

Řešení

1. Setřídění pole nevyžaduje ne více než $\frac{1}{2}n(n-1)$ porovnání, tj. složitost třídy $O(n^2)$.
2. Porovnání všech dvojic sousedních prvků bude vyžadovat $n-1$ porovnání, tj. složitost třídy $O(n)$.

Celková složitost algoritmu je tedy $O(\max(n^2, n)) = O(n^2)$.

Využití limit k výpočtům

Rychlost růstu funkcí lze snadněji počítat pomocí limit:

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & t(n) \text{ roste pomaleji než } g(n) \\ c & t(n) \text{ roste stejně rychle jako } g(n) \\ \infty & t(n) \text{ roste rychleji než } g(n) \end{cases}$$

Je zřejmé, že:

$t(n) \in O(g(n)) \quad \Leftrightarrow \quad t(n)$ roste pomaleji nebo stejně rychle než $g(n)$

$t(n) \in \Omega(g(n)) \quad \Leftrightarrow \quad t(n)$ roste stejně rychle nebo rychleji než $g(n)$

$t(n) \in \Theta(g(n)) \quad \Leftrightarrow \quad t(n)$ roste stejně rychle jako $g(n)$

Využití limit k výpočtům (pokrač.)

Některé užitečné vzorce

L'Hospitalovo pravidlo

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{t'(n)}{g'(n)}$$

Stirlingův vzorec

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Využití limit k výpočtům – příklad I

Srovnejte rychlost růstu funkcí $\frac{1}{2}n(n-1)$ a n^2.

$$\begin{aligned}
\lim_{n\to\infty} \frac{\frac{1}{2}n(n-1)}{n^2} &= \frac{1}{2}\lim_{n\to\infty}\frac{n^2-n}{n^2} \\
&= \frac{1}{2}\lim_{n\to\infty}\left(1-\frac{1}{n}\right) \\
&= \frac{1}{2}\left(\lim_{n\to\infty}1-\lim_{n\to\infty}\frac{1}{n}\right) \\
&= \frac{1}{2}(1-0) = \frac{1}{2} > 0
\end{aligned}$$

Funkce $\frac{1}{2}n(n-1)$ a n^2 rostou stejně rychle, tedy

$$\frac{1}{2}n(n-1) \in \Theta(n^2)$$

.

Využití limit k výpočtům – příklad II

Srovnejte rychlost růstu funkcí $\log_2 n$ a $\sqrt{n}$.

$$\begin{aligned}
\lim_{n\to\infty} \frac{\log_2 n}{\sqrt{n}} &= \lim_{n\to\infty} \frac{\left(\log_2 n\right)'}{\left(\sqrt{n}\right)'} \\
&= \lim_{n\to\infty} \frac{(\log_2 e)\frac{1}{n}}{\frac{1}{2\sqrt{n}}} = (\log_2 e) \lim_{n\to\infty} \frac{\frac{1}{n}}{\frac{1}{2\sqrt{n}}} \\
&= \log_2 e \lim_{n\to\infty} \frac{2\sqrt{n}}{n} \\
&= 2\log_2 e \lim_{n\to\infty} \frac{1}{\sqrt{n}} = 0
\end{aligned}$$

Funkce $\log_2 n$ tedy **roste pomaleji** než $\sqrt{n}$.

Využití limit k výpočtům – příklad III

Srovnejte rychlost růstu funkcí $n!$ a 2^n.

$$\begin{aligned}\lim_{n\to\infty}\frac{n!}{2^n} &= \lim_{n\to\infty}\frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{2^n}\\ &= \sqrt{2\pi}\lim_{n\to\infty}\sqrt{n}\frac{n^n}{2^n e^n}\\ &= \sqrt{2\pi}\lim_{n\to\infty}\sqrt{n}\left(\frac{n}{2e}\right)^n = \infty\end{aligned}$$

Poznámky

- Funkce $n!$ tedy roste rychleji než 2^n.
- Definice Θ-notace nevylučuje, že $n! \in \Omega(2^n)$, ale výpočet podle limity jasně říká, že $n!$ roste rychleji než 2^n

Základní třídy složitosti

Přestože teoreticky existuje nekonečně mnoho tříd složitosti, složitost většiny algoritmů padne do několika málo tříd.

Třída	Jméno	Poznámka
1	konstantní	složitost nezávisí na velikosti vstupu; jen velmi málo algoritmů
$\log n$	logaritmická	typicky algoritmy redukující velikost vstupu konstantním faktorem; vyhledávání půlením intervalu
n	lineární	algoritmy zpracovávající seznam o n prvcích; např. sekvenční vyhledávání
$n \log n$	lineárně-logaritmická	„rozděl a panuj“ algoritmy; průměrné složitosti QuickSortu, MergeSortu

Základní třídy složitosti (pokrač.)

Třída	Jméno	Poznámka
n^2	kvadratická	obecně algoritmy se dvěma vnořenými cykly; elementární metody třídění, sčítání matic typu $n \times n$
n^3	kubická	obecně algoritmy se třemi vnořenými cykly; násobení matic typu $n \times n$
2^n	exponenciální	typicky generování všech podmnožin n prvkové množiny
$n!$	faktoriál	typicky generování všech permutací n prvkové množiny

Vliv multiplikativní konstanty

- Třída složitosti je dána až na multiplikativní konstantu, která obvykle není přesně specifikována.
- Mohl by tedy algoritmus s vyšší třídy složitosti běžet, pro nějaké rozumné n, rychleji než algoritmus z lepší třídy? Například:

Algoritmus	Doba běhu
A	n^3
B	$10^6 n^2$

A bude lepší než B pro $n < 10^6$.

- Multiplikativní konstanty obvykle nabývají podobných, relativně malých, hodnot.
- **Lze očekávat, že algoritmy s nižší složitostí budou lepší než ty vyšší složitostí už pro středně velké vstupy.**

Zdroje pro samostatné studium

- Kniha [2], kapitola 2.2, strany 52 – 61
- Kniha [3], kapitoly 3.1 a 3.2, strany 49 – 63

Analýza složitosti algoritmů

Analýza nerekurzivních algoritmů

Nalezení největšího prvku v poli n čísel

Vstup : Pole $A[0 \ldots n-1]$ celých čísel
Výstup: Největší prvek pole A

```
1 max ← A[0];
2 for i ← 1 to n - 1 do
3 |   if A[i] > max then
4 |   |   max ← A[i];
5 |   end
6 end
7 return max;
```

Nalezení největšího prvku v poli n čísel (pokrač.)

Pracovní postup

1. Velikost vstupu – velikost pole n
2. Základní operace:
   - nejčastěji vykonávané operace jsou uvnitř cyklu – porovnání $A[i] > max$ a přiřazení $max \leftarrow A[i]$
   - základní operací bude **porovnání**, protože se
     - provede v každém průchodu cyklem,
     - je to klíčová operace pro algoritmus, „Kolik dvojic prvků musím porovnat, abych našel maximum?“
3. Počet porovnání je stejný pro všechny vstupy velikosti n, není nutné rozlišovat mezi nejlepším, průměrným a nejhorším případem

Nalezení největšího prvku v poli n čísel (pokrač.)

4. Počet základních operací, porovnání, $C(n)$ bude roven

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

5. **Závěr**: Nalezení největšího prvku v poli n čísel je **lineární algoritmus**.

Nalezení největšího prvku v poli n čísel, všechny operace

Počet operací	Popis
1	přiřazení $max \leftarrow A[0]$
1	přiřazení $i \leftarrow 1$
$n - 1$	porovnání $i \leq n - 1$
$n - 1$	zvýšení i o 1
$n - 1$	porovnání $A[i] > max$
$n - 1$	přiřazení $max \leftarrow A[i]$
1	vrácení výsledku **return** max
$4(n-1)+3 = 4n - 1 \in \Theta(n)$	

Závěr: Nalezení největšího prvku v poli n čísel je **lineární algoritmus**.

Obecný postup určení časové složitosti nerek. algoritmů

1. Volba parametru, či parametrů, reprezentujícího velikost vstupu n.
2. Nalezení základních operací algoritmu (jsou to ty v nejvíce vnořeném cyklu!).
3. Závisí počet základních operací jen na velikosti vstupu? Pokud závisí i na něčem dalším, musíme zkoumat nejhorší, nejlepší a průměrný případ zvlášť.
4. Sestavení vztahu, resp. vztahů, („vzorečků“) vyjadřujících počet, resp. počty, provedení základních operací.
5. Zjednodušení sestavených vztahů a, nebo aspoň, stanovení řádového růstu.

Užitečné součtové vzorce

$$\sum (a_i \pm b_i) = \sum a_i \pm \sum b_i \tag{3}$$

$$\sum c a_i = c \sum a_i \tag{4}$$

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{m} a_i + \sum_{i=m+1}^{n} a_i \tag{5}$$

$$\sum_{i=l}^{u} 1 = 1 + 1 + \cdots + 1 = u - l + 1 \tag{6}$$

Konkrétně

$$\sum_{i=1}^{n} 1 = n \in \Theta(n) \tag{7}$$

Užitečné součtové vzorce (pokrač.)

$$\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{1}{2}n(n+1) \approx \frac{1}{2}n^2 \in \Theta(n^2) \tag{8}$$

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \cdots + n^2 = \frac{1}{6}n(n+1)(2n+1) \approx \frac{1}{3}n^3 \in \Theta(n^3) \tag{9}$$

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1}, \text{ pro } a \neq 1 \tag{10}$$

Konkrétně

$$\sum_{i=0}^{n} 2^i = 2^0 + 2^1 + \cdots + 2^n = 2^{n+1} - 1 \in \Theta(2^n) \tag{11}$$

Unikátnost prvků v poli

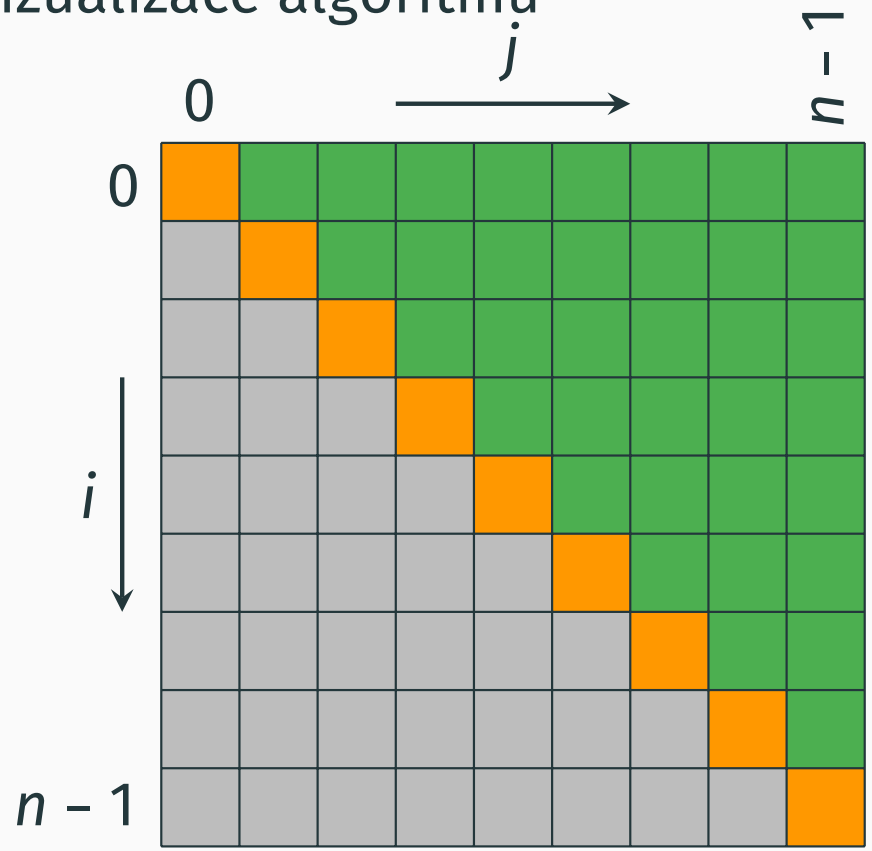
Je dáno pole o n prvcích. Naším úkolem provést analýzu algoritmu, který zjistí, zda všechny prvky v poli jsou navzájem různé, čili unikátní.

Vstup : Pole $A[0 \ldots n-1]$
Výstup: Vrací true, pokud jsou všechny prvky unikátní, jinak vrací false

```
1 for i ← 0 to n − 2 do
2 |   for j ← i + 1 to n − 1 do
3 |   |   if A[i] = A[j] then
4 |   |   |   return false;
5 |   |   end
6 |   end
7 end
8 return true;
```

Unikátnost prvků v poli (pokrač.)

Vizualizace algoritmu



Legenda

dvojice, které **je nutné** otestovat

prvek sám se sebou není nutné testovat

dvojice testované v předchozích průchodech cyklem

Unikátnost prvků v poli (pokrač.)

Pracovní postup

1. Velikost vstupu – velikost pole n
2. Základní operace – nejvíce vnořený cyklus obsahuje jedinou operaci, porovnání $A[i] = A[j]$
3. Závislost pouze na n? Ne, počet zákl. operací závisí i na tom, zda se v poli objeví shodný prvek. Tudíž provádíme analýzu **nejhoršího**, nejlepšího a průměrného případu.
4. Sestavení vztahů. Pro nejhorší případ je z vnitřního cyklu je patrné, že nesmí dojít k předčasnému ukončení cyklu, a to buď:
   - 4.1 protože všechny prvky jsou unikátní nebo
   - 4.2 je shodná až poslední dvojice.

Unikátnost prvků v poli (pokrač.)

Tudíž provedeme:

- jedno porovnání pro každý průchod vnitřním cyklem, tj. $j = i + 1, \ldots, n - 1$
- vnější cyklus, v každém svém průchodu, zopakuje celý vnitřní cyklus , tj. $i = 0, \ldots, n - 2$

Unikátnost prvků v poli (pokrač.)

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 \qquad \text{podle (6)}$$

$$= \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i \qquad \text{podle (3)}$$

$$= (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2} \qquad \text{podle (4) a (8)}$$

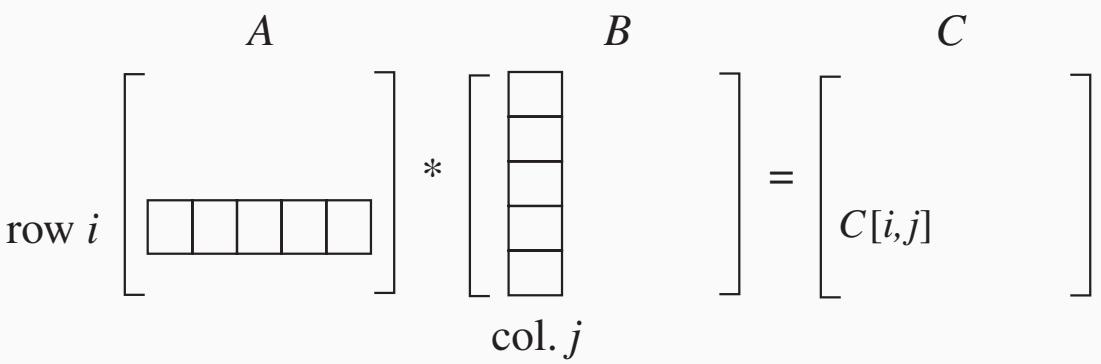
$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} \qquad \text{podle (6)}$$

$$= \frac{1}{2} n(n-1) \approx \frac{1}{2} n^2 \in \Theta(n^2)$$

Násobení čtvercových matic

Naším úkolem je provést analýzu algoritmu pro výpočet součinu $C = AB$ dvou čtvercových matic A a B řádu n.

Z definice jsou prvky matice rovny skalárním součinům řádků matice A se sloupci matice B.



$$c_{i,j} = \sum_{k=0}^{n-1} a_{i,k} b_{k,j}$$

pro všechna
$0 \leq i, j \leq n - 1$

$$C[i,j] = A[i,0] \times B[0,j] + \cdots + A[i,k] \times B[k,j] + \cdots + A[i,n-1] \times B[n-1,j]$$

Násobení čtvercových matic (pokrač.)

Vstup : Dvě čtvercové matice A a B řádu n
Výstup: Čtvercová matice C řádu n, kde $C = AB$

```
1 for i ← 0 to n – 1 do
2 |   for j ← 0 to n – 1 do
3 |   |   C[i, j] ← 0;
4 |   |   for k ← 0 to n – 1 do
5 |   |   |   C[i, j] ← C[i, j] + A[i, k] × B[k, j];
6 |   |   end
7 |   end
8 end
9 return C;
```

Násobení čtvercových matic (pokrač.)

Pracovní postup

1. Velikost vstupu – řád matice n
2. Základní operace:
   - nejvíce vnořený cyklus obsahuje dvě operace – sčítání a násobení,
   - v každém průchodu se obě provedou přesně jedenkrát,
   - historická tradice velí počítat násobení (bývalo mnohokrát pomalejší než sčítání),
   - zavedeme $M(n)$ jako celkový počet násobení.
3. Počet operací závisí pouze na n – nejhorší, nejlepší a průměrný případ splývají

Násobení čtvercových matic (pokrač.)

4. Sestavení vztahů – počet násobení v nejvnitřnějším cyklu

$$\sum_{k=0}^{n-1} 1$$

Celkový počet násobení je roven

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1$$

Násobení čtvercových matic (pokrač.)

Pomocí vztahů (4) a (6) postupně dostáváme

$$
\begin{aligned}
M(n) &= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} (n - 1 - 0 + 1) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = n \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1 \\
&= n \sum_{i=0}^{n-1} (n - 1 - 0 + 1) = n \sum_{i=0}^{n-1} n = n^2 \sum_{i=0}^{n-1} 1 \\
&= n^2 (n - 1 - 0 + 1) \\
&= n^3
\end{aligned}
$$

Násobení čtvercových matic (pokrač.)

Neformální postup

1. algoritmus musí vypočítat $n \times n$ prvků matice C
2. každý prvek matice C je vypočten jako skalární součin i-tého řádku matice A a j-tého sloupce matice B
3. řádky i sloupce mají n prvků, které musíme vynásobit
4. celkem tedy $n^2 \times n = n^3$ násobení

Násobení čtvercových matic (pokrač.)

Doba běhu algoritmu na konkrétním počítači

$$T(n) \approx c_m M(n) = c_m n^3$$

započteme-li i sčítání

$$T(n) \approx c_m M(n) + c_a A(n) = c_m n^3 + c_a n^3 = (c_m + c_a) n^3,$$

kde c_m resp. c_a je doba trvání násobení resp. sčítání, $A(n)$ je počet operací sčítání, platí $A(n) = M(n)$.

Shrnutí

Doba běhu algoritmu se může, v závislosti na konkrétním počítači, měnit, ale **řádová složitost algoritmu (n^3) zůstává**.

Počet bitů v binárním zápisu čísla

Naším úkolem je analyzovat algoritmus, který pro dané přirozené číslo n vypočte počet bitů nutných pro zápis čísla n v binární soustavě.

Vstup : Přirozené číslo n

Výstup: Počet bitů v binárním zápisu čísla n

```
1 count ← 1;
2 while n > 1 do
3 |   count ← count + 1;
4 |   n ← ⌊n/2⌋;
5 end
6 return count;
```

- Velikost vstupu – jedno číslo?

Počet bitů v binárním zápisu čísla (pokrač.)

- Základní operace – sčítání, dělení, porovnání s 1?
- Nejdůležitější je, v tomto případě, určit počet průchodů cyklem. Počet porovnání je o jedna větší než počet průchodů cyklem.
- Hodnota čísla n se v každém průchodu cyklem zmenšuje na polovinu, což vede ke vztahu

$$\lfloor \log_2 n \rfloor + 1$$

  a což odpovídá vztahu (2).
- K odvození budeme potřebovat umět řešit rekurzivní rovnice...

Zdroje pro samostatné studium

- Kniha [2], kapitola 2.3, strany 61 – 70

Analýza složitosti algoritmů

Analýza rekurzivních algoritmů

Výpočet faktoriálu

Naším úkolem je analyzovat rekurzivní algoritmus, který pro dané přirozené číslo n vypočte jeho faktoriál $n!$.

$$n! = \begin{cases} 1 & \text{pro } n = 0 \\ n(n-1)! & \text{jinak} \end{cases}$$

```
1 Function F(n)
  |  Vstup: Přirozené číslo n
  |  Výsledek: Hodnota n!
2 |  if n = 0 then
3 |  |  return 1;
4 |  end
5 |  return n × F(n − 1);
6 end
```

Výpočet faktoriálu (pokrač.)

- Velikost vstupu – jedno číslo. Budeme brát v úvahu počet bitů? Ne, bylo by to komplikované.
- Základní operace – násobení. Alternativně lze uvažovat počet porovnání $n = 0$, které odpovídá počtu volání funkce F.
- Bude nás zajímat počet násobení $M(n)$ v závislosti na čísle n

Výpočet faktoriálu (pokrač.)

- Pro $n > 0$ se funkce $F(n)$ počítá jako

$$F(n) = F(n-1) \times n$$

a odtud

$$M(n) = M(n-1) + 1, \tag{12}$$

kde
$M(n-1)$ výpočet funkce $F(n-1)$ a
1 vynásobení výsledku $F(n-1)$ číslem n.

- Hodnota $M(n)$ není definována explicitně tj. jako funkce n například n^3, ale **implicitně** pomocí vztahu založeném na hodnotě totožné funkce pro jiné přirozené číslo, konkrétně $n-1$. Jde o tzv. **rekurentní (rekurzivní) vztah**.

Výpočet faktoriálu (pokrač.)

Poznámka

Explicitní	Implicitní
výslovný, přímý, jasný, zřetelný	zahrnutý, obsažený, ale nevyjádřený přímo
otevřeně, přímo vyjádřený, nenechávající nic zamlčeného, skrytého	nikoli zjevný, samo sebou se rozumějící
významově navzájem opačná slova	

Příklad

Explicitně: Lžete.
Implicitně: O pravdivosti vašeho tvrzení by se dalo s úspěchem pochybovat.

Výpočet faktoriálu (pokrač.)

- Cílem je najít najít explicitní vyjádření $M(n)$.
- Vztah $M(n) = M(n-1) + 1$ není jednoznačný, pro jednoznačné řešení je nutné definovat **počáteční podmínku**.
- Podmínka

```
if n = 0 then
    return 1;
end
```

  nám říká:

  1. nejmenší n pro které se algoritmus provede je $n = 0$ a
  2. v tomto případě algoritmus neprovede žádné násobení, tudíž $M(0) = 0$.

Výpočet faktoriálu (pokrač.)

- Celkově tedy pro výpočet $M(n)$ platí

$$\begin{aligned} M(n) &= M(n-1)+1 \text{ pro } n > 0 \\ M(0) &= 0 \end{aligned}$$

- Soustavu budeme řešit **metodou zpětné substituce**. Do

$$M(n) = M(n-1)+1$$

dosadíme za $M(n-1) = M(n-2)+1$

$$M(n) = [M(n-2)+1]+1 = M(n-2)+2$$

Výpočet faktoriálu (pokrač.)

a opět dosadíme za $M(n-2) = M(n-3) + 1$

$$M(n) = [M(n-3) + 1] + 2 = M(n-3) + 3.$$

Je zřejmé, že

$$M(n) = M(n-i) + i$$

Poznámka

Správnost této formule lze dokázat pomocí matematické indukce.

Výpočet faktoriálu (pokrač.)

Počáteční podmínka je definována pro $n = 0$, takže musíme dosadit za $i = n$ a dostáváme

$$M(n) = M(n-1) + 1 = \cdots = M(n-i) + i = \cdots = M(n-n) + n = n.$$

Shrnutí

1. Výsledek $M(n) = n$ byl víceméně očekávaný.
2. Iterativní algoritmus provede stejný počet násobení jak rekurzivní navíc bez režie spojené se zásobníkem pro volání funkcí.
3. Důležitý je ale popsaný přístup jak řešit rekurentní rovnice.

Obecný postup určení časové složitosti rekurzivních algoritmů

1. Volba parametru, či parametrů, reprezentujícího velikost vstupu n.
2. Nalezení základních operací algoritmu.
3. Závisí počet základních operací jen na velikosti vstupu? Pokud závisí i na něčem dalším, musíme zkoumat nejhorší, nejlepší a průměrný případ zvlášť.
4. Sestavení rekurentního vztahu a vhodné počáteční podmínky, vyjadřující počet provedení základních operací.
5. Zjednodušení sestavených vztahů a, nebo aspoň, stanovení řádového růstu.

Hanojské věže (Tower of Hanoi)

- Matematický hlavolam, autor Édouard Lucas, 1883.
- Hlavolam se skládá ze tří tyčí.
- Na začátku je na jedné tyči nasazeno několik kotoučů různých poloměrů, seřazených od největšího (vespod) po nejmenší (nahoře).

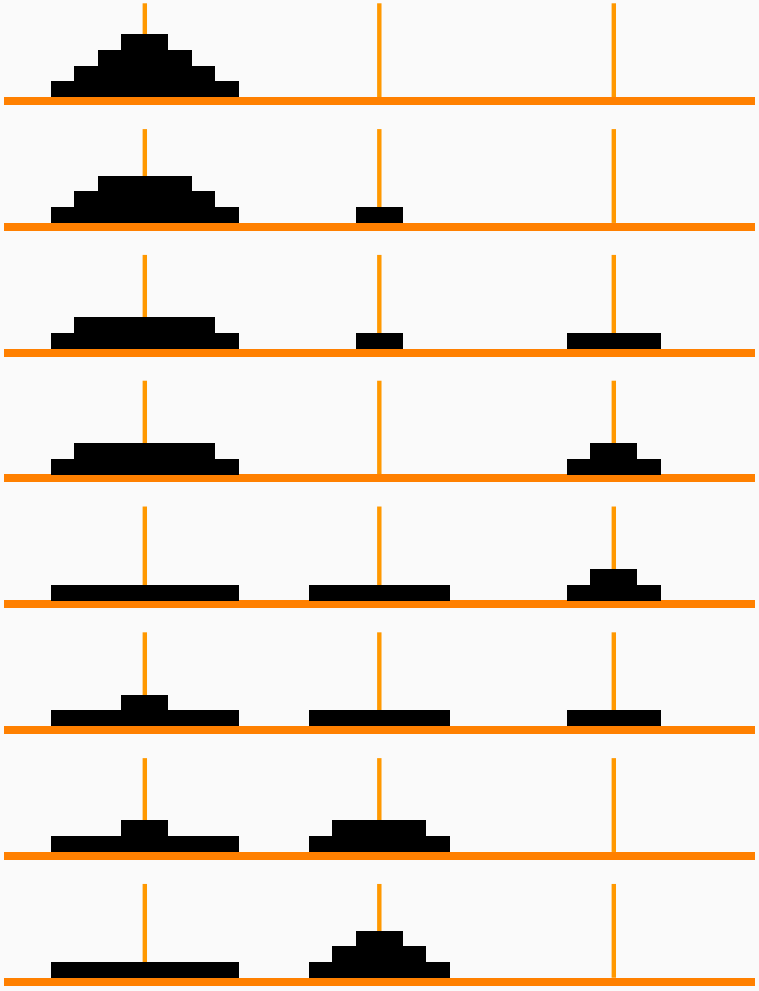
- Úkolem je přemístit všechny kotouče z první tyče na třetí tyč za pomoci druhé tyče.
- Pravidla hry:
  - V jednom tahu lze přemístit jen jeden kotouč.

Hanojské věže (Tower of Hanoi) (pokrač.)

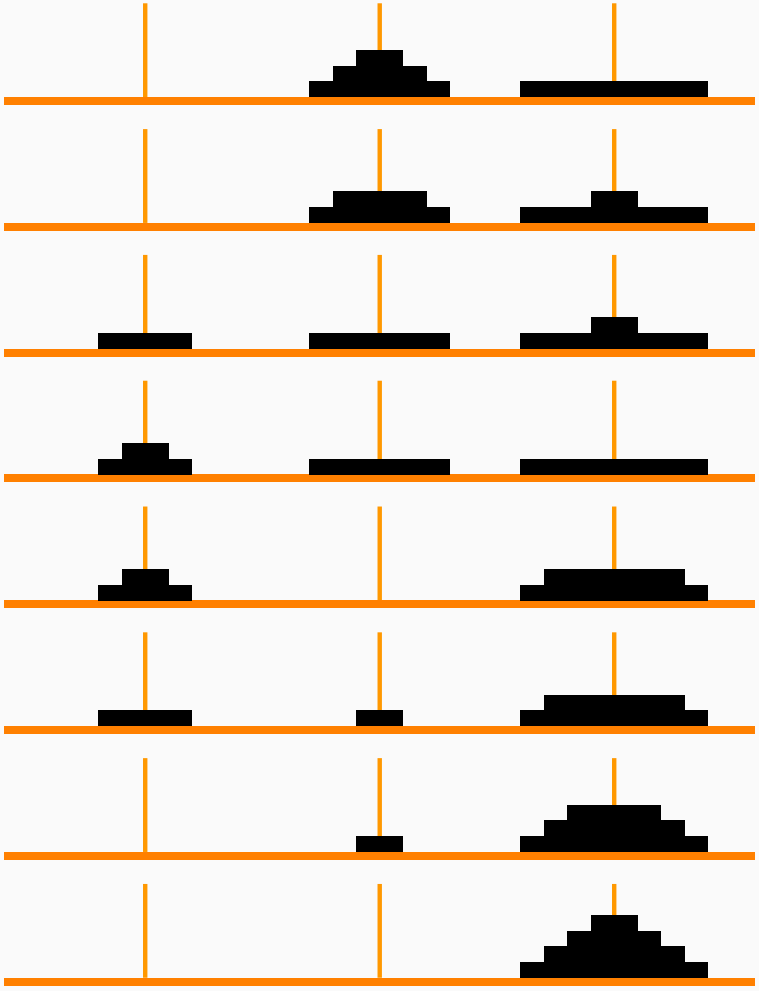
  - Jeden tah se sestává ze sejmutí vrchního kotouče z některé tyče a jeho navlečení na jinou tyč.
  - Je zakázáno položit větší kotouč na menší.
- Podle legendy stojí v Hanoji klášter, v němž jsou hanojské věže se 64 zlatými kotouči. Mniši každý den v poledne přemístí jeden kotouč. V okamžiku, kdy bude přemístěn poslední kotouč, nastane konec světa.
- Hlavně klid! Řešení tohoto hlavolamu pro 64 kotoučů vyžaduje $2^{64} - 1 = 18\,446744073709551615$ tahů. I kdyby přemístili každou sekundu jeden kotouč (a postupovali nejkratším možným způsobem), doba řešení je cca 600 miliard let.

Hanojské věže (Tower of Hanoi) (pokrač.)

Kroky řešení 1 – 8



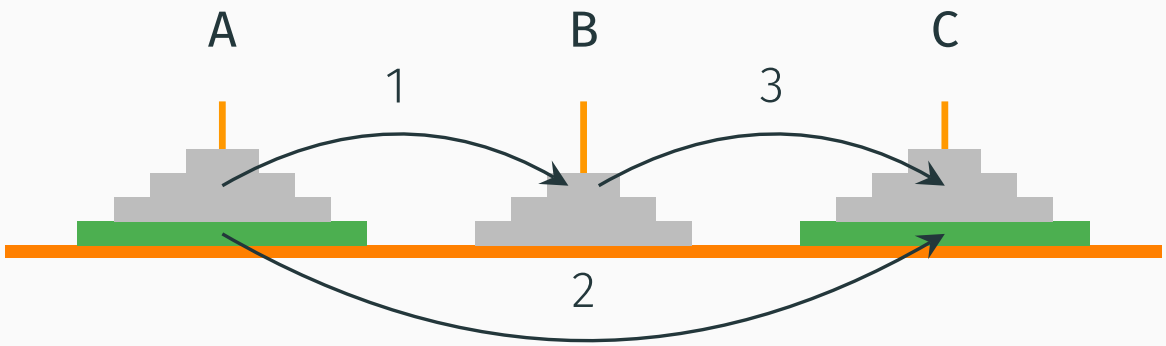
Kroky řešení 9 – 16



Hanojské věže (Tower of Hanoi) (pokrač.)

Řešení problému pro n disků

1. Přesun $n - 1$ disků z tyče A na tyče B (pomocí tyče C).
2. Přesun největšího disku z tyče A přímo na tyč C.
3. Přesun $n - 1$ disků z tyče B na tyč C (pomocí tyče A).



Pro $n = 1$ existuje triviální řešení...

Hanojské věže (Tower of Hanoi) (pokrač.)

Pracovní postup

1. Velikost vstupu – počet disků n.
2. Základní operace – přesun disku.
3. Závislost pouze na n? Ano. Tudíž **nemusíme** zkoumat nejhorší, nejlepší a průměrný případ zvlášť.
4. Sestavení vztahů

$$M(n) = \begin{cases} 1 & \text{pro } n = 1 \\ 2M(n-1)+1 & \text{jinak} \end{cases} \tag{13}$$

Hanojské věže (Tower of Hanoi) (pokrač.)

5. Řešení metodou zpětné substituce. Do vztahu

$$M(n) = 2M(n-1) + 1$$

dosadíme $M(n-1) = 2M(n-2) + 1$

$$M(n) = 2[2M(n-2) + 1] + 1 = 2^2 M(n-2) + 2 + 1,$$

opět dosadíme $M(n-2) = 2M(n-3) + 1$

$$M(n) = 2^2[2M(n-3) + 1] + 2 + 1 = 2^3 M(n-3) + 2^2 + 2 + 1,$$

po dalším dosazení dostaneme

$$M(n) = 2^4 M(n-4) + 2^3 + 2^2 + 2 + 1$$

Hanojské věže (Tower of Hanoi) (pokrač.)

Po i-tém dosazení dostáváme

$$\begin{aligned} M(n) &= 2^i M(n-i) + \underbrace{2^{i-1} + 2^{i-2} + \cdots + 2 + 1}_{\text{sečteme podle (11)}} \\ &= 2^i M(n-i) + 2^i - 1. \end{aligned}$$

Počáteční podmínky platné pro $n = 1$ dosáhneme při $i = n - 1$

$$\begin{aligned} M(n) &= 2^{n-1} M(n-(n-1)) + 2^{n-1} - 1 \\ &= 2^{n-1} M(1) + 2^{n-1} - 1 \\ &= 2^{n-1} \cdot 1 + 2^{n-1} - 1 = 2 \cdot 2^{n-1} - 1 = 2^{n-1+1} - 1 \\ &= 2^n - 1 \end{aligned}$$

Hanojské věže (Tower of Hanoi) (pokrač.)

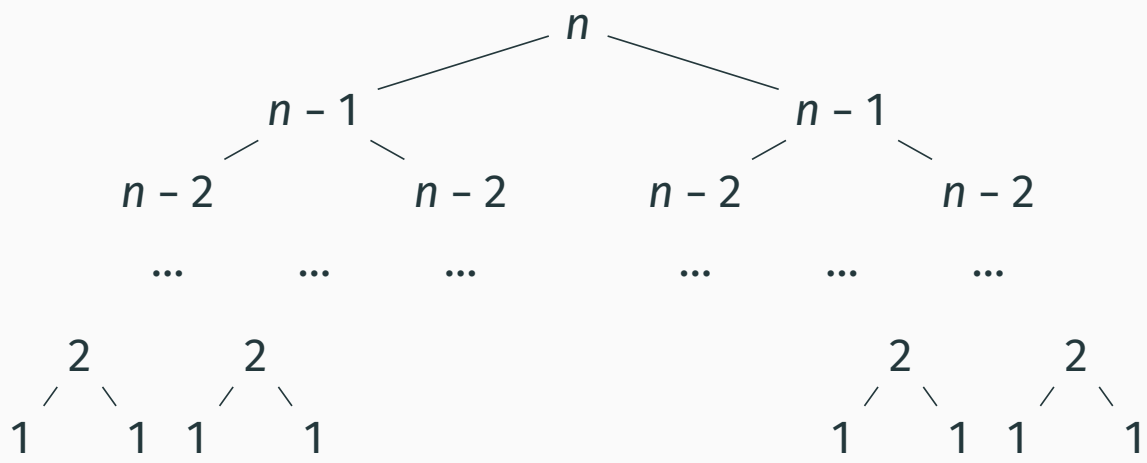
6. Závěr:
   - 6.1 Navržený rekurzivní algoritmus provede exponenciální počet základních operací vzhledem k velikosti vstupu.
   - 6.2 Algoritmus je použitelný jen pro malá n, což není způsobeno nevhodným návrhem. Je to způsobeno podstatou problému – lze dokázat, že toto je nejlepší možný algoritmus.

Opatrně s rekurzivními algoritmy

Obecně je nutné k rekurzivním algoritmům přistupovat velice opatrně, protože jejich stručnost může maskovat neefektivitu.

Hanojské věže (Tower of Hanoi) (pokrač.)

Vizualizace rekurzivního volání



Počet uzlů odpovídá počtu volání rekurzivní funkce

$$C(n) = \sum_{l=0}^{n-1} 2^l = 2^n - 1,$$

kde l je číslo úrovně ve stromu.

Zdroje pro samostatné studium

- Kniha [2], kapitola 2.4, strany 70 – 79

Děkuji za pozornost