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 (daná hlavně počtem provedených instrukcí) může být pro různé algoritmy různá.
- Algoritmy bychom chtěli mezi sebou porovnávat a zvolit si ten lepší.
- 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

Vezměme si nějaký konkrétní stroj vykonávající nějaký algoritmus — např. stroj RAM, Turingův stroj, ...

Budeme předpokládat, že pro daný stroj $\mathcal{M}$ máme nějak definované následující dvě funkce, které každému vstupu z množiny všech vstupů In přiřazují:

- $time_{\mathcal{M}} : In \to \mathbb{N} \cup \{\infty\}$ — **dobu výpočtu** stroje $\mathcal{M}$ nad daným vstupem
- $space_{\mathcal{M}} : In \to \mathbb{N} \cup \{\infty\}$ — **množství paměti** použité strojem $\mathcal{M}$ při výpočtu nad daným vstupem

Poznámka: Toto ještě není časová a paměťová složitost algoritmu vykonávaného daným strojem $\mathcal{M}$.

Složitost algoritmu — Turingovy stroje

Například v případě **Turingových strojů** nás bude zajímat následující:

- **počet kroků**, které daný stroj vykoná při výpočtu nad daným slovem

  — tato hodnota reprezentuje dobu výpočtu

- **množství políček pásky**, které daný stroj při výpočtu nad daným slovem navštíví

  — tato hodnota reprezentuje množství použité paměti

Složitost algoritmu — Turingovy stroje

Formálně můžeme tyto pojmy reprezentovat následovně (předpokládáme daný Turingův stroj $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, F)$):

- Funkce

$$time_{\mathcal{M}} : \Sigma^* \to \mathbb{N} \cup \{\infty\}$$

Pro $w \in \Sigma^*$ hodnota $time_{\mathcal{M}}(w)$ udává počet kroků, které stroj $\mathcal{M}$ vykoná při výpočtu nad vstupem w.

Tj., pokud je tento výpočet konečný a vypadá následovně

$$\alpha_0 \longrightarrow \alpha_1 \longrightarrow \alpha_2 \longrightarrow \alpha_3 \longrightarrow \quad \cdots \quad \longrightarrow \alpha_{t-1} \longrightarrow \alpha_t$$

kde α_t je koncová konfigurace, je $time_{\mathcal{M}}(w) = t$.

V případě nekonečného výpočtu

$$\alpha_0 \longrightarrow \alpha_1 \longrightarrow \alpha_2 \longrightarrow \alpha_3 \longrightarrow \quad \cdots$$

je $time_{\mathcal{M}}(w) = \infty$.

Složitost algoritmu — Turingovy stroje

- Funkce

$$space_{\mathcal{M}} : \Sigma^* \to \mathbb{N} \cup \{\infty\}$$

Pro $w \in \Sigma^*$ hodnota $space_{\mathcal{M}}(w)$ udává počet políček pásky, které stroj $\mathcal{M}$ během výpočtu nad vstupem w navštíví.

Poznámka: Je zřejmé, že v případě konečného výpočtu je i hodnota $space_{\mathcal{M}}(w)$ konečná.

V případě nekonečného výpočtu může být hodnota $space_{\mathcal{M}}(w)$ buď konečná nebo nekonečná.

Složitost algoritmu — stroje RAM

Dobu výpočtu **stroje RAM** je možno počítat dvěma různými způsoby:

- **jednotková míra** — počet provedených instrukcí
- **logaritmická míra** — součet doby trvání jednotlivých instrukcí; doba trvání každé instrukce závisí na počtu bitů hodnot, se kterými se v této instrukci pracuje.

  Například:

  - Doba provádění instrukcí sčítání a odčítání je rovna součtu počtů bitů jejich operandů.
  - Doba provádění instrukcí násobení a dělení je rovna součinu počtů bitů jejich operandů.
  - Doba provádění instrukce přístupu do paměti (load, store) je dána jako součet počtu bitů adresy a počtu bitů čteného či zapisovaného čísla.

  Poznámka: Při počítání počtu bitů daného čísla se počítá, že hodnota 0 má 1 bit.

Složitost algoritmu — stroje RAM

Rovněž množství paměti použité během výpočtu strojem RAM je možno počítat dvěma různými způsoby:

- **jednotková míra** — množství použitých buněk paměti, tj. počet buněk, do kterých stroj během výpočtu zapisoval nebo z nich četl.
- **logaritmická míra** — maximální množství paměti, které bylo během výpočtu potřeba pro nějakou konfiguraci.

  Množství paměti pro konfiguraci je dáno jako součet počtu bitů všech použitých buněk paměti a počtu bitů jejich adres.

Složitost algoritmu — stroje RAM

Jednotková míra realisticky odráží množství provedené práce či množství použité paměti pouze v případech, kdy jsou hodnoty čísel uložených v buňkách paměti „malé“, tj. pokud by je ve skutečné implementaci pro „rozumně“ velké vstupy bylo možné reprezentovat např. 32-bitovými či 64-bitovými čísly.

Složitost algoritmu — stroje RAM

- U strojů, které nemají instrukci násobení, se dá snadno ukázat, že každá jednotlivá instrukce může vytvořit číslo, které má nanejvýš o jeden bit více než (v absolutní hodnotě) větší z jejích operandů.

  U takovýchto strojů obsahuje po t krocích výpočtu každá buňka číslo, které má nejvýše $t + m + n$ bitů, kde m je počet bitů největší konstanty, která se vyskutuje v programu a n je největší počet bitů, jaké má libovolné číslo na vstupu.

- Pokud má stroj instrukci násobení, může nějaká buňka paměti obsahovat po t krocích číslo, které má až zhruba 2^t bitů.

Velikost vstupu

Pro různé vstupy provede program různý počet instrukcí.

Pokud chceme počet provedených instrukcí nějak analyzovat, je vhodné si zavést pojem **velikost vstupu**.

Typicky je velikost vstupu číslo, které udává, jak je daná instance „velká“ (čím větší číslo, tím větší instance).

Poznámka: Velikost vstupu si v daném konkrétním případě můžeme definovat, jak chceme a jak je to pro další analýzu výhodné.

Co přesně zvolíme jako velikost vstupu není předem dáno, ale z podstaty zadaného problému většinou nějak přirozeně vyplývá, co za velikost vstupu zvolit.

Velikost vstupu

Příklady:

- Pro problém „Třídění“, kde vstupem je sekvence čísel $a_1, a_2, \ldots, a_n$ a výstupem jsou tato čísla setříděná, můžeme vzít jako velikost vstupu hodnotu n.
- Pro problém „Prvočíselnost“, kde vstupem je přirozené číslo x, a kde se ptáme, zda x je prvočíslo, můžeme vzít jako velikost vstupu počet bitů čísla x.

  (Jinou možností by bylo vzít jako velikost vstupu přímo hodnotu x.)

Velikost vstupu

Někdy je vhodné popsat velikost vstupu pomocí více čísel.

Například u problémů, kde vstupem je graf, můžeme definovat velikost vstupu jako dvojici čísel n, m, kde:

- n – počet vrcholů grafu
- m – počet hran grafu

Poznámka: Jinou možností by bylo definovat velikost vstupu jako jediné číslo $n + m$.

Velikost vstupu

Obecně můžeme pro libovolný problém definovat velikost vstupu následovně:

- Pokud je vstupem slovo w z nějaké obecedy Σ:
  délka slova w
- Pokud je vstupem sekvence bitů (tj. slovo z abecedy $\{0, 1\}$):
  počet bitů v této sekvenci
- Pokud je vstupem přirozené číslo x:
  počet bitů nutných k zápisu čísla x

Časová složitost

Chceme analyzovat konkrétní algoritmus (jeho konkrétní implementaci).

Zajímá nás, kolik instrukcí se provede, pokud algoritmus dostane vstup velikosti $0, 1, 2, 3, 4, \ldots$.

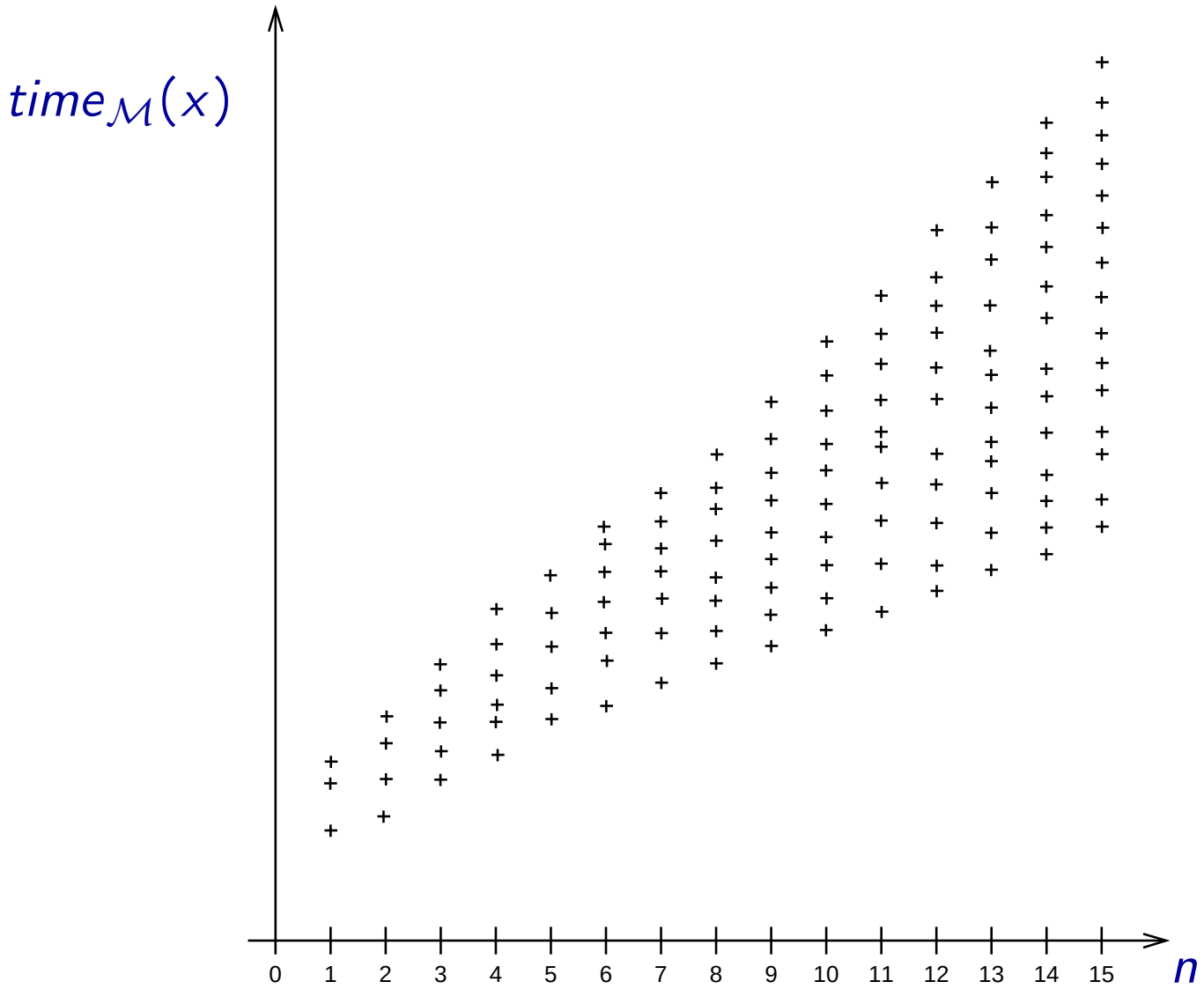
Je zřejmé, že i pro vstupy, které mají stejnou velikost, může být počet provedených instrukcí různý.

Označme si velikost vstupu $x \in \mathit{In}$ jako $\mathit{size}(x)$.

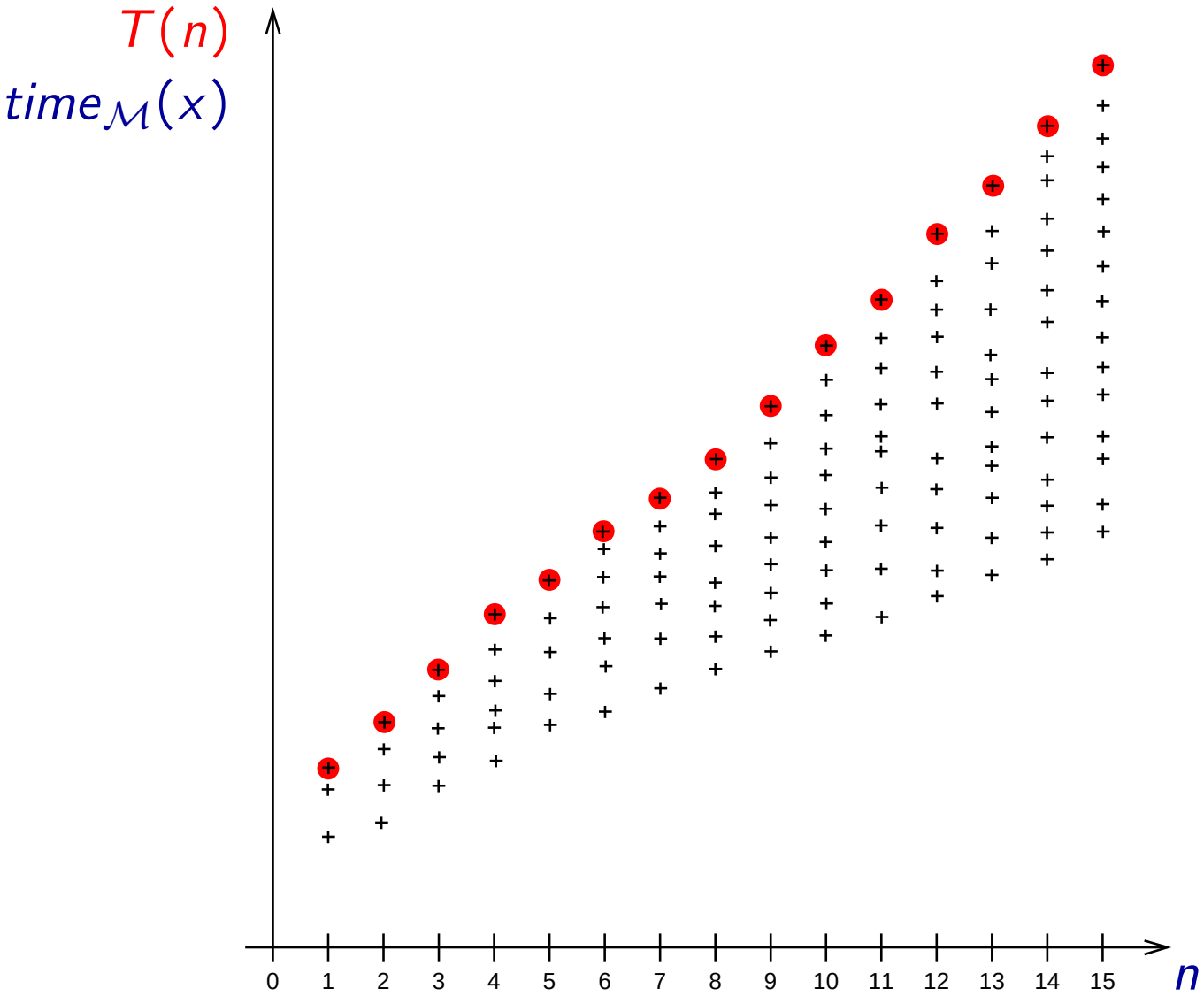
Nyní definujme následující funkci $T : \mathbb{N} \to \mathbb{N}$ takovou, že pro $n \in \mathbb{N}$ je

$$T(n) = \max\{\, \mathit{time}_{\mathcal{M}}(x) \mid x \in \mathit{In},\ \mathit{size}(x) = n \,\}$$

Časová složitost v nejhorším případě



Časová složitost v nejhorším případě



Časová a prostorová složitost v nejhorším případě

Takto definované funkci $T(n)$ (tj. funkci, která pro daný algoritmus a danou definici velikosti vstupu přiřazuje každému přirozenému číslu n maximální počet instrukcí, které algoritmus provede, pokud dostane vstup velikosti n) se říká **časová složitost algorimu v nejhorším případě**.

$$T(n) = \max\{\, time_{\mathcal{M}}(x) \mid x \in In,\ size(x) = n \,\}$$

Analogicky můžeme definovat **prostorovou (paměťovou) složitost algoritmu v nejhorším případě** jako funkci $S(n)$, kde:

$$S(n) = \max\{\, space_{\mathcal{M}}(x) \mid x \in In,\ size(x) = n \,\}$$

Časová složitost v průměrném případě

Kromě časové složitosti v nejhorším případě má smysl zkoumat i časovou složitost **v průměrném případě**.

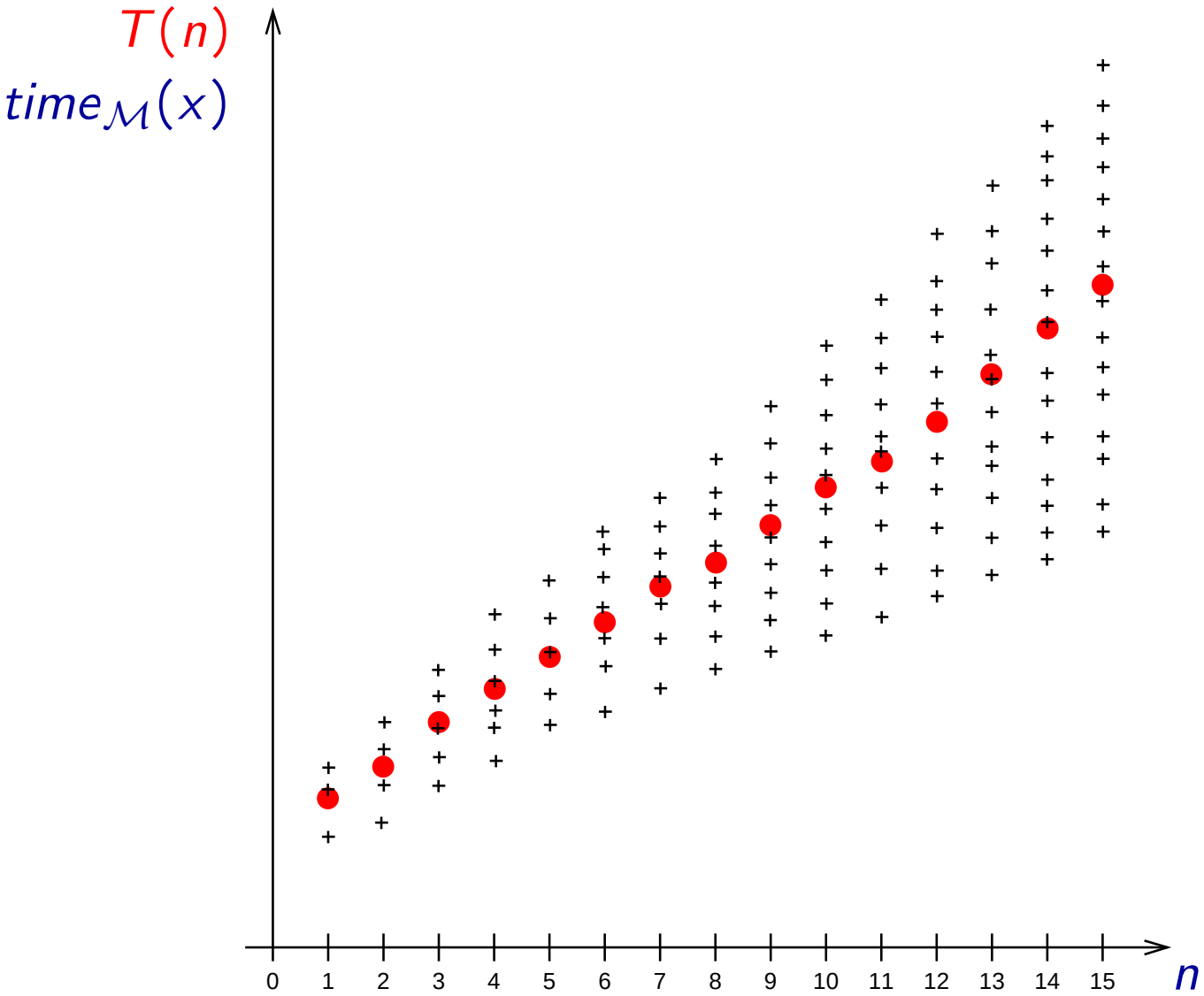
V tomto případě $T(n)$ nedefinujeme jako maximum, ale jako aritmetický průměr z hodnot

$$\{ time_{\mathcal{M}}(x) \mid x \in In, size(x) = n \}$$

- Určit časovou složitost v průměrném případě je většinou těžší než určit časovou složitost v nejhorším případě.
- Často se tyto dvě funkce příliš neliší, někdy je ale rozdíl významný.

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

Časová složitost v průměrném případě



Rychlost růstu funkcí

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

	n							
$T(n)$	20	40	60	80	100	200	500	1000
n	$20\,\mu$s	$40\,\mu$s	$60\,\mu$s	$80\,\mu$s	0.1 ms	0.2 ms	0.5 ms	1 ms
$n \log n$	$86\,\mu$s	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 dni
2^n	1.05 s	12.75 dni	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×
$T_2(n) = n^3$	10^5	10×
$T_3(n) = 2^n$	50	+10

Asymptotická notace

- Přesnou složitost bývá problém vyjádřit.
- Přesná složitost je silně závislá na konkrétním zvoleném modelu a konkrétní implementaci (na detailech této implementace).
- Složitost nás většinou zajímá hlavně pro velké vstupy. Pro malé vstupy obvykle i neefektivní algoritmus proběhne rychle.
- Ve většině případů nepotřebujeme znát přesný počet provedených instrukcí, ale spokojíme se s odhadem toho, jak rychle tento počet narůstá se zvětšováním velikosti vstupu.
- Proto zavádíme tzv. **asymptotickou notaci**, která nám umožní zanedbat méně důležité detaily a odhadnout přibližně, jak rychle daná funkce roste, a která analýzu podstatným způsobem zjednodušuje.

Asymptotická notace

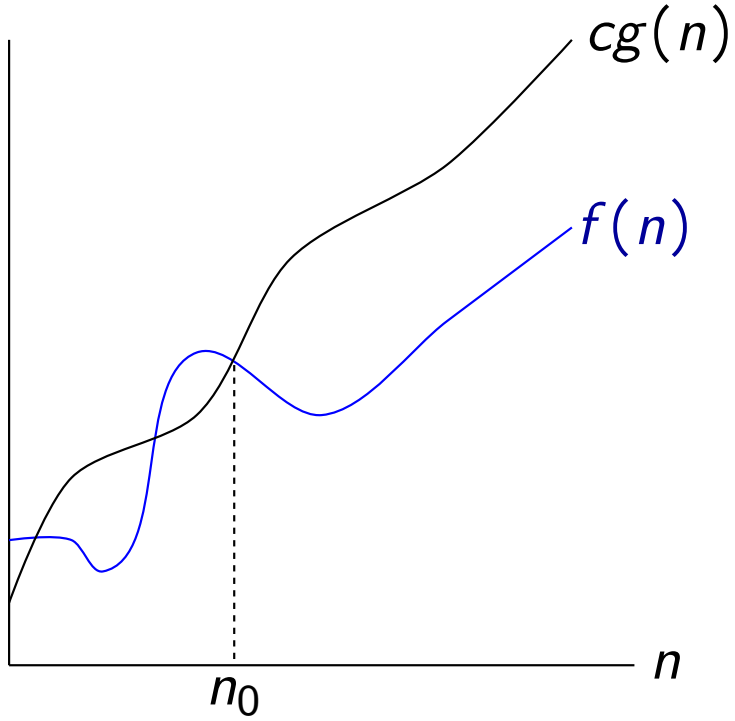
Vezměme si libovolnou funkci $g : \mathbb{N} \to \mathbb{N}$. Zápisy $\mathcal{O}(g)$, $\Omega(g)$, $\Theta(g)$, $o(g)$ a $\omega(g)$ označují **množiny funkcí** typu $\mathbb{N} \to \mathbb{N}$, kde:

- $\mathcal{O}(g)$ – množina všech funkcí, které rostou nejvýše tak rychle jako g
- $\Omega(g)$ – množina všech funkcí, které rostou alespoň tak rychle jako g
- $\Theta(g)$ – množina všech funkcí, které rostou stejně rychle jako g
- $o(g)$ – množina všech funkcí, které rostou pomaleji než funkce g
- $\omega(g)$ – množina všech funkcí, které rostou rychleji než funkce g

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

- $\mathcal{O}$ – velké „O“
- Ω – velké řecké písmeno „omega“
- Θ – velké řecké písmeno „theta“
- o – malé „o“
- ω – malé „omega“

Asymptotická notace – symbol $\mathcal{O}$

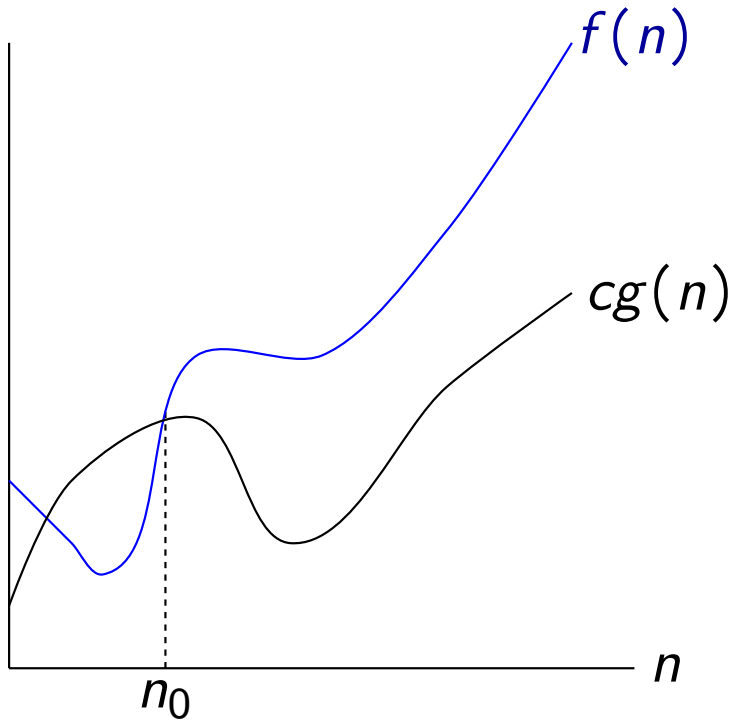


Definice

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

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

Asymptotická notace – symbol Ω

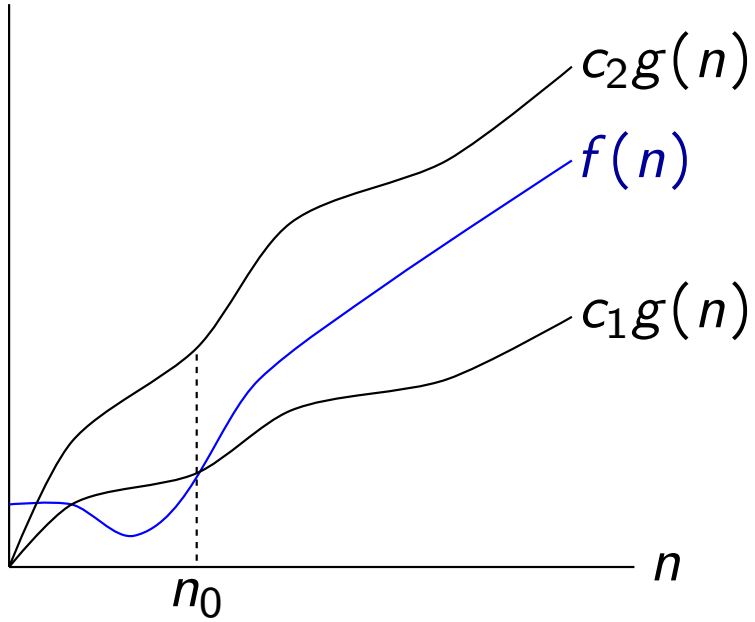


Definice

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

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

Asymptotická notace – symbol Θ



Definice

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

$$f \in \mathcal{O}(g) \quad \text{a} \quad f \in \Omega(g).$$

Asymptotická notace – symboly o and ω

Definice

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

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = 0$$

Definice

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

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = +\infty$$

Asymptotická notace

Pro jednoduchost uvažujeme v předchozích definicích pouze funkce typu $\mathbb{N} \to \mathbb{N}$.

Ve skutečnosti by se tyto definice daly rozšířit na všechny **asymptoticky nezáporné** funkce typu $\mathbb{R}_+ \to \mathbb{R}$, které navíc mohou být na nějakém konečném podintervalu svého definičního oboru nedefinované.

Funkce $f : \mathbb{R}_+ \to \mathbb{R}$ je **asymptoticky nezáporná** pokud pro ni platí:

$$(\exists n_0 \geq 0)(\forall n \geq n_0)(f(n) \geq 0)$$

Poznámka: Pro $n < n_0$, může být hodnota $f(n)$ nedefinovaná.

$\mathbb{R}_+ = \{x \in \mathbb{R} \mid x \geq 0\}$

Asymptotická notace

- Existují dvojice funkcí $f, g : \mathbb{N} \to \mathbb{N}$ takové, že

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

  například

$$f(n) = n \qquad g(n) = \begin{cases} n^2 & \text{pokud } n \bmod 2 = 0 \\ \lceil \log_2 n \rceil & \text{jinak} \end{cases}.$$

Asymptotická notace

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

- 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 $f(n) \in \mathcal{O}(n^k)$ pro nějaké $k > 0$
  exponenciální, pokud $f(n) \in \mathcal{O}(c^{n^k})$ pro nějaké $c > 1$ a $k > 0$

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

Asymptotická notace

Jak bylo uvedeno, výrazy $\mathcal{O}(g)$, $\Omega(g)$, $\Theta(g)$, $o(g)$ a $\omega(g)$ označují určité množiny funkcí.

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

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

Tato konvence se používá zejména v zápisu rovnic nebo nerovnic.

Příklad: $3n^3 + 5n^2 - 11n + 2 = 3n^3 + \mathcal{O}(n^2)$

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

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 x (tj. pro vstup x se instrukce I_i provede m_i krát), tak celkový počet instrukcí provedených pro vstup x je
$$m_1 + m_2 + \cdots + m_k.$$
- Vezměme si funkce $t_1, t_2, \ldots, t_k$, kde $t_i : \mathbb{N} \to \mathbb{N}$, přičemž $t_i(n)$ je maximum z počtu provedení instrukce I_i pro všechny vstupy velikosti n.
- Zjevně platí, že pro libovolnou funkci t_i je $T \in \Omega(t_i)$.
- Zjevně také platí $T \in \mathcal{O}(t_1 + t_2 + \cdots + t_k)$.

Složitost algoritmů

- Připomeňme si, že pokud $f \in \mathcal{O}(g)$, pak $f + g \in \mathcal{O}(g)$.
- Pokud tedy pro některou fukci t_i platí, že pro všechny t_j, kde $j \neq i$, je $t_j \in \mathcal{O}(t_i)$, pak

$$T \in \mathcal{O}(t_i).$$

- 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 $t_i(n)$ krát), protože platí

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

Složitost algoritmů

Pokusme se analyzovat časovou složitost následujícího algoritmu:

Algoritmus: Třídění přímým vkládáním

```
INSERTION-SORT (A, n):
    for j := 1 to n − 1 do
        x := A[j]
        i := j − 1
        while i ≥ 0 and A[i] > x do
            A[i + 1] := A[i]
            i := i − 1
        A[i + 1] := x
```

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 krát.
- Existují vstupy, pro které platí že pro každou hodnotu j od 1 do $(n-1)$ se vnitřní cyklus **while** provede právě j krát.
- V nejhorším případě se tedy cyklus **while** provede celkem m krát, kde
$$m = 1 + 2 + \cdots + (n-1) = (1 + (n-1)) \cdot \frac{n-1}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$$
- 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 $T \in \mathcal{O}(n^2)$.

- 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 $T \in \Omega(n^2)$.

Složitost algoritmů

Při používání asymptotických odhadů časové složitosti algoritmů bychom si měli být vědomi některých úskalí:

- Asyptotické odhady se týkají pouze toho, jak roste čas s rostoucí velikostí vstupu.
- Neříkají nic o konkrétní době výpočtu. V asymptotické notaci mohou být skryty velké konstanty.
- Algoritmus, který má lepší asymptotickou časovou složitost než nějaký jiný algoritmus, může být ve skutečnosti rychlejší až pro nějaké hodně velké vstupy.
- Většinou analyzujeme složitost v nejhorším případě. Pro některé algoritmy může být doba výpočtu v nejhorším případě mnohem větší než doba výpočtu na „typických“ instancích.

Složitost algoritmů

- Můžeme si to ilustrovat na algoritmech pro třídění.

Algoritmus	Nejhorší případ	Průměrný případ
Bubblesort	$\Theta(n^2)$	$\Theta(n^2)$
Heapsort	$\Theta(n \log n)$	$\Theta(n \log n)$
Quicksort	$\Theta(n^2)$	$\Theta(n \log n)$

- Quicksort má horší asymptotickou složitost v nejhorším případě než Heapsort, stejnou asymptotickou složitost v průměrném případě a přesto je v praxi nejrychlejší.

Prostorová (paměťová) složitost algoritmů

- Zatím jsme se zajímali o čas, který potřebujeme k výpočtu
- Někdy bývá kritickou velikost paměti potřebné k provedení výpočtu.

Připomeňme, že pro stroj $\mathcal{M}$ hodnota $space_{\mathcal{M}}(x)$ udává množství paměti použité strojem $\mathcal{M}$ při výpočtu nad vstupem x.

Definice

Pro daný stroj $\mathcal{M}$ je **prostorová (paměťová) složitost** stroje $\mathcal{M}$ funkce $S : \mathbb{N} \to \mathbb{N}$, kde

$$S(n) = \max\{\, space_{\mathcal{M}}(x) \mid x \in In, size(x) = n \,\}$$

Prostorová (paměťová) složitost algoritmů

- Pro konkrétní problém můžeme mít dva algoritmy takové, že jeden má menší prostorovou složitost a druhý zase časovou složitost.
- Je-li časová složitost algoritmu v $\mathcal{O}(f(n))$ je i prostorová v $\mathcal{O}(f(n))$.

Složitost algoritmů

Orientační typické hodnoty velikosti vstupu n, pro které algoritmus s danou časovou složitostí ještě většinou zvládne na „běžném PC“ spočítat výsledek ve zlomku sekundy nebo maximálně v řádu sekund.

(Závisí to samozřejmě výrazně na konkrétních detailech. Navíc se zde předpokládá, že v asymptotické notaci nejsou skryty nějaké velké konstanty.)

$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$
$1\,000\,000 - 100\,000\,000$	$100\,000 - 1\,000\,000$	$1000 - 10\,000$	$100 - 1000$

$2^{\mathcal{O}(n)}$	$\mathcal{O}(n!)$
$20 - 30$	$10 - 15$

Polynomiální algoritmy

Polynom — funkce tvaru

$$a_k n^k + a_{k-1} n^{k-1} + \cdots + a_2 n^2 + a_1 n + a_0$$

kde $a_0, a_1, \ldots, a_k$ jsou konstanty.

Příklady polynomů:

$$4n^3 - 2n^2 + 8n + 13 \qquad 2n + 1 \qquad n^{100}$$

Polynomiální algoritmy

Funkce f je **polynomiální**, jestliže je shora omezena nějakým polynomem, tj. jestliže existuje nějaká konstanta k taková, že $f \in \mathcal{O}(n^k)$.

Polynomiální jsou například funkce, které patří do následujících tříd:

$$\mathcal{O}(n) \qquad \mathcal{O}(n \log n) \qquad \mathcal{O}(n^2) \qquad \mathcal{O}(n^5) \qquad \mathcal{O}(\sqrt{n}) \qquad \mathcal{O}(n^{100})$$

Funkce jako 2^n nebo $n!$ polynomiální nejsou — pro libovolně velkou konstantu k platí

$$2^n \in \omega(n^k) \qquad\qquad n! \in \omega(n^k)$$

Polynomiální algoritmy

Polynomiální algoritmus — algoritmus, jehož časová složitost je polynomiální — tj. shora omezená nějakým polynomem (tedy v $\mathcal{O}(n^k)$, kde k je nějaká konstanta).

Na pojem „polynomiální algoritmus“ se lze dívat jako na určitou aproximaci toho, jaké algoritmy jsou obecně vnímány jako „efektivní“ a prakticky použitelné i pro poměrně velké vstupy.

Zhruba se dá říct, že:

- Polynomiální algoritmy jsou efektivní algoritmy, které se dají prakticky použít i pro relativně velké vstupy.
- Algoritmy, které polynomiální nejsou (tj. mají vyšší časovou složitost než polynomiální, např. exponenciální), obecně za efektivní považovány nejsou.

  Takovéto algoritmy jsou většinou použitelné jen pro „malé“ vstupy.

Polynomiální algoritmy

Jsou zde některá „ale“:

- Algoritmus, který má časovou složitost třeba v $\Theta(n^{100})$, určitě nelze považovat z praktického hlediska za efektivní.
- Dá se ukázat, že pro libovolné k je možné uměle sestrojit příklad algoritmického problému, který je možné vyřešit algoritmem s časovou složitostí v $\mathcal{O}(n^{k+1})$, ale přitom neexistuje žádný algoritmus s časovou složitostí v $\mathcal{O}(n^{k})$, který by ho řešil.
- U „přirozeně“ definovaných problémů, které se řeší v praxi, to nebývá tak, že by pro ně existoval polynomiální algoritmus s nějakým vysokým stupněm polynomu a přitom neexistoval algoritmus s nízkým stupněm polynomu.

  Většinou nastává u takových problémů jedna ze dvou možností:

  - Je znám polynomiální algoritmus, kde stupeň polynomu je poměrně nízký, např. nejvýše 5.
  - Pro daný problém není znám žádný polynomiální algoritmus.

Polynomiální algoritmy

- Z praktického hlediska může být někdy i algoritmus se složitostí třeba $\Theta(n^2)$ považován pro některé účely za neefektivní — např. pro hodně velké vstupy nebo pokud máme nějaká silná omezení ohledně doby výpočtu.
- Naopak pro některé účely může být i algoritmus s exponenciální časovou složitostí v praxi použitelný.
- Existují příklady algoritmů, které mají sice exponenciální časovou složitost v nejhorším případě, ale pro mnoho vstupů ve skutečnosti pracují efektivně a jsou schopny v rozumném čase zpracovat i poměrně velké vstupy.

Složitost algoritmů

Pro většinu běžných algoritmických problémů nastává jedna ze tří možností:

- Je znám polynomiální algoritmus se složitostí $\mathcal{O}(n^k)$, kde k je nějaké velmi malé číslo (např. 5 a častěji třeba 3 a méně).

- Není znám žádný polynomiální algoritmus a nejlepší známé algoritmy mají složitosti jako třeba $2^{\Theta(n)}$, $\Theta(n!)$ nebo nějaké ještě větší.

  V některých případech může být znám i důkaz, že pro daný problém žádný polynomiální algoritmus neexistuje (tj. nedá se vytvořit).

- Není znám žádný algoritmus, který řeší daný problém (a případně je i dokázáno, že žádný takový algoritmus neexistuje).

Složitost algoritmů

Typický příklad polynomiálního algoritmu — násobení matic s časovou složitostí $\Theta(n^3)$ a paměťovou složitostí $\Theta(n^2)$:

Algoritmus: Násobení matic

```
MATRIX-MULT(A, B, C, n):
    for i := 1 to n do
        for j := 1 to n do
            x := 0
            for k := 1 to n do
                x := x + A[i][k] * B[k][j]
            C[i][j] := x
```

Složitost algoritmů

- Při hrubé analýze složitosti často stačí spočítat počet do sebe vnořených smyček — a tento počet pak udává stupeň polynomu

  Příklad: Tři vnořené cykly při násobení matic — časová složitost algoritmu je $\mathcal{O}(n^3)$.

- Pokud neprobíhají všechny smyčky např. od 0 do n, ale počet průchodů vnitřními smyčkami se při různých iteracích vnější smyčky mění, podrobnější analýza může být komplikovanější.

  Většinou to pak vede na počítání součtů různých typů číselných řad (např. aritmetické, geometrické, apod.).

  Často dá taková podrobnější analýza podobný výsledek jako hrubá analýza, mnohdy však může být složitost zjištěná touto podrobnější analýzou podstatně nižší než by vyplývalo z hrubého odhadu.

Aritmetická posloupnost

Aritmetická posloupnost — číselná řada $a_0, a_1, \ldots, a_{n-1}$, kde

$$a_i = a_0 + i \cdot d,$$

kde d je nějaká konstanta nezávislá na i.

V aritmetické posloupnosti tedy pro všechna i platí $a_{i+1} = a_i + d$.

Příklad: Aritmetická posloupnost, kde $a_0 = 1$, $d = 1$ a $n = 100$:

$$1, 2, 3, 4, 5, 6, \ldots, 96, 97, 98, 99, 100$$

Součet aritmetické posloupnosti:

$$\sum_{i=0}^{n-1} a_i = a_0 + a_1 + \cdots + a_{n-1} = \frac{1}{2} n (a_0 + a_{n-1})$$

Aritmetická posloupnost

Příklad:

$$1 + 2 + \cdots + n \;=\; \frac{1}{2}n(n+1) \;=\; \frac{1}{2}n^2 + \frac{1}{2}n \;=\; \Theta(n^2)$$

Konkrétně například pro $n = 100$ je

$$1 + 2 + \cdots + 100 \;=\; 50 \cdot 101 \;=\; 5050.$$

Aritmetická posloupnost

Důkaz: Označme

$$s = \sum_{i=0}^{n-1} a_i = a_0 + a_1 + \cdots + a_{n-1}$$

$$\begin{aligned}
2s &= s + s \\
&= (a_0 + a_1 + \cdots + a_{n-1}) + (a_0 + a_1 + \cdots + a_{n-1}) \\
&= (a_0 + a_1 + \cdots + a_{n-1}) + (a_{n-1} + a_{n-2} + \cdots + a_0) \\
&= (a_0 + a_{n-1}) + (a_1 + a_{n-2}) + \cdots + (a_{n-1} + a_0) \\
&= ((a_0 + 0 \cdot d) + (a_0 + (n-1) \cdot d)) + ((a_0 + 1 \cdot d) + (a_0 + (n-2) \cdot d)) + \\
&\quad \cdots + ((a_0 + (n-1) \cdot d) + (a_0 + 0 \cdot d)) \\
&= n \cdot (a_0 + a_0 + (n-1) \cdot d) \\
&= n \cdot (a_0 + a_{n-1})
\end{aligned}$$

Aritmetická posloupnost

Příklad: $s = 1 + 2 + 3 + \cdots + 99 + 100$

$$\begin{aligned} 2s &= s + s \\ &= (1 + 2 + \cdots + 100) + (1 + 2 + \cdots + 100) \\ &= (1 + 2 + \cdots + 100) + (100 + 99 + \cdots + 1) \\ &= (1 + 100) + (2 + 99) + (3 + 98) + \cdots + (99 + 2) + (100 + 1) \\ &= 100 \cdot (1 + 100) = 10100 \end{aligned}$$

Takže

$$s = \frac{1}{2} \cdot 10100 = 5050$$

Geometrická posloupnost

Geometrická posloupnost — číselná řada $a_0, a_1, \ldots, a_n$, kde

$$a_i = a_0 \cdot q^i,$$

kde q je nějaká konstanta nezávistlá na i.

V geometrické posloupnosti tedy pro všechna i platí $a_{i+1} = a_i \cdot q$.

Příklad: Geometrická posloupnost, kde $a_0 = 1$, $q = 2$ a $n = 14$:

1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384

Součet geometrické posloupnosti (kde $q \neq 1$):

$$\sum_{i=0}^{n} a_i = a_0 + a_1 + \cdots + a_n = a_0 \frac{q^{n+1} - 1}{q - 1}$$

Geometrická posloupnost

Příklad:

$$1 + q + q^2 + \cdots + q^n = \frac{q^{n+1} - 1}{q - 1}$$

Speciálně pro $q = 2$:

$$1 + 2^1 + 2^2 + 2^3 + \cdots + 2^n = \frac{2^{n+1} - 1}{2 - 1} = 2 \cdot 2^n - 1 = \Theta(2^n)$$

Geometrická posloupnost

Důkaz: Označme

$$s = \sum_{i=0}^{n} a_i = a_0 + a_1 + \cdots + a_n$$

$$\begin{aligned}
s &= a_0 \cdot q^0 + a_0 \cdot q^1 + \cdots + a_0 \cdot q^n \\
s \cdot q &= (a_0 \cdot q^0 + a_0 \cdot q^1 + \cdots + a_0 \cdot q^n) \cdot q \\
&= a_0 \cdot q^1 + a_0 \cdot q^2 + \cdots + a_0 \cdot q^{n+1} \\
s \cdot q - s &= a_0 \cdot q^{n+1} - a_0 \cdot q^0 \\
s \cdot (q-1) &= a_0 \cdot (q^{n+1} - 1) \\
s &= a_0 \cdot \frac{q^{n+1} - 1}{q - 1}
\end{aligned}$$

Složitost algoritmů

Exponenciální funkce: funkce tvaru c^n, kde c je konstanta — např. funkce 2^n

Logaritmus — inverzní funkce k exponenciální funkci: pro dané n je

$$\log_c n$$

taková hodnota x, že $c^x = n$.

Složitost algoritmů

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024
11	2048
12	4096
13	8192
14	16384
15	32768
16	65536
17	131072
18	262144
19	524288
20	1048576

n	$\lceil \log_2 n \rceil$
0	—
1	0
2	1
3	2
4	2
5	3
6	3
7	3
8	3
9	4
10	4
11	4
12	4
13	4
14	4
15	4
16	4
17	5
18	5
19	5
20	5

n	$\log_2 n$
1	0
2	1
4	2
8	3
16	4
32	5
64	6
128	7
256	8
512	9
1024	10
2048	11
4096	12
8192	13
16384	14
32768	15
65536	16
131072	17
262144	18
524288	19
1048576	20

Složitost algoritmů

Příklady toho, kde se při analýze algoritmů objevují exponenciální funkce a logaritmy:

- Nějaká hodnota se opakovaně zmenšuje na polovinu nebo naopak zdvojnásobuje.

  Například u **binárního vyhledávání** (metodou půlení intervalu) se s každou iterací cyklu zmenšuje velikost intervalu na polovinu.

  Předpokládejme, že pole má velikost n.

  Jaká je minimální velikost pole n, při které se provede alespoň k iterací?

  Odpověď: 2^k

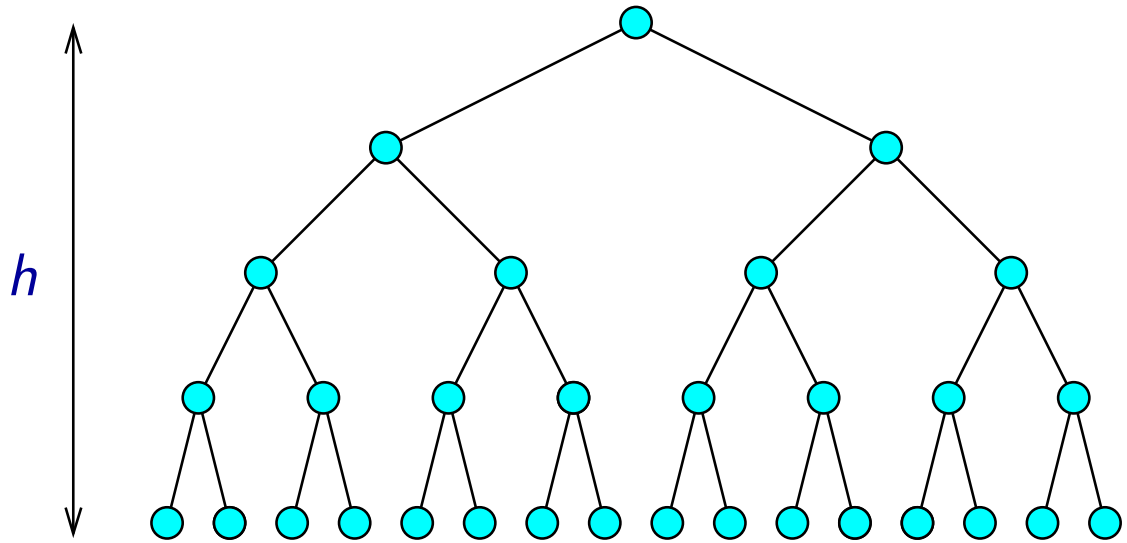
  Platí tedy $k = \log_2(n)$. Časová složitost algoritmu je pak $\Theta(\log n)$.

Složitost algoritmů

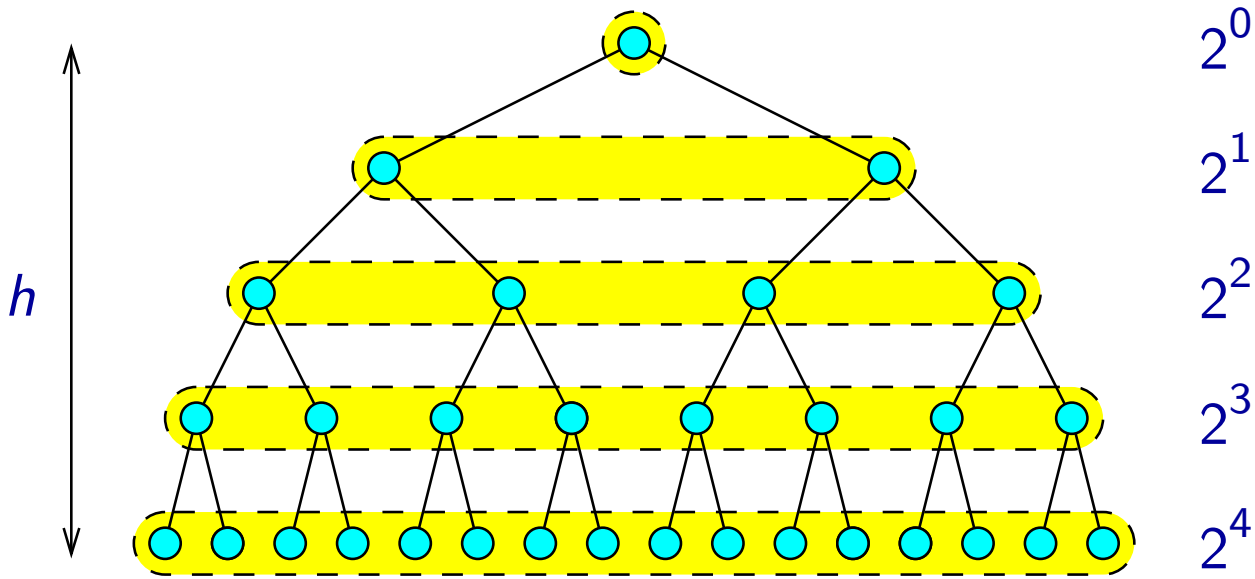
- Pomocí n bitů je možno reprezentovat čísla od 0 do $2^n - 1$.
- Minimální počet bitů potřebných pro uložení přirozeného čísla x reprezentovaného binárně je
$$\lceil \log_2(x+1) \rceil.$$
- Dokonale vyvážený binární strom o výšce h má $2^{h+1} - 1$ vrcholů, z čehož 2^h jsou listy.
- Dokonale vyvážený binární strom o n vrcholech má výšku zhruba $\log_2 n$.

  Ilustrační příklad: Kdybychom nakreslili vyvážený strom o $n = 1\,000\,000$ vrcholech tak, aby sousední vrcholy byly vzdáleny o 1 cm a výška každé vrstvy byla také 1 cm, měl by tento strom na šířku 10 km a na výšku zhruba 20 cm.

Dokonale vyvážený binární strom výšky h:

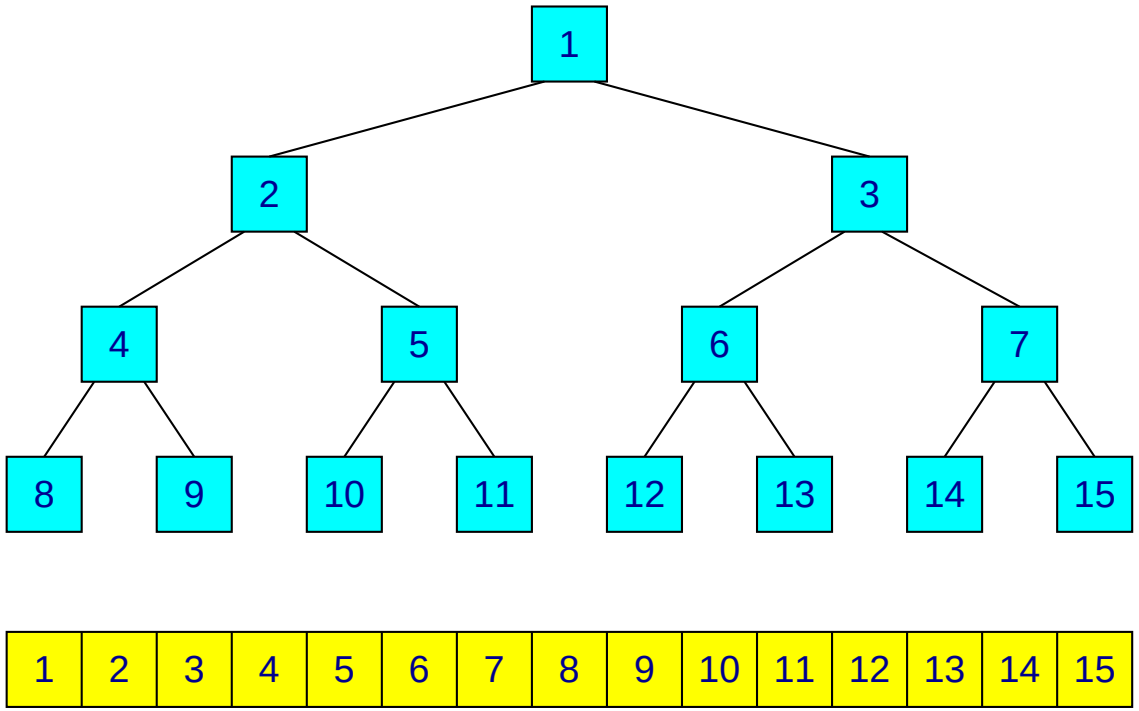


Dokonale vyvážený binární strom výšky h:



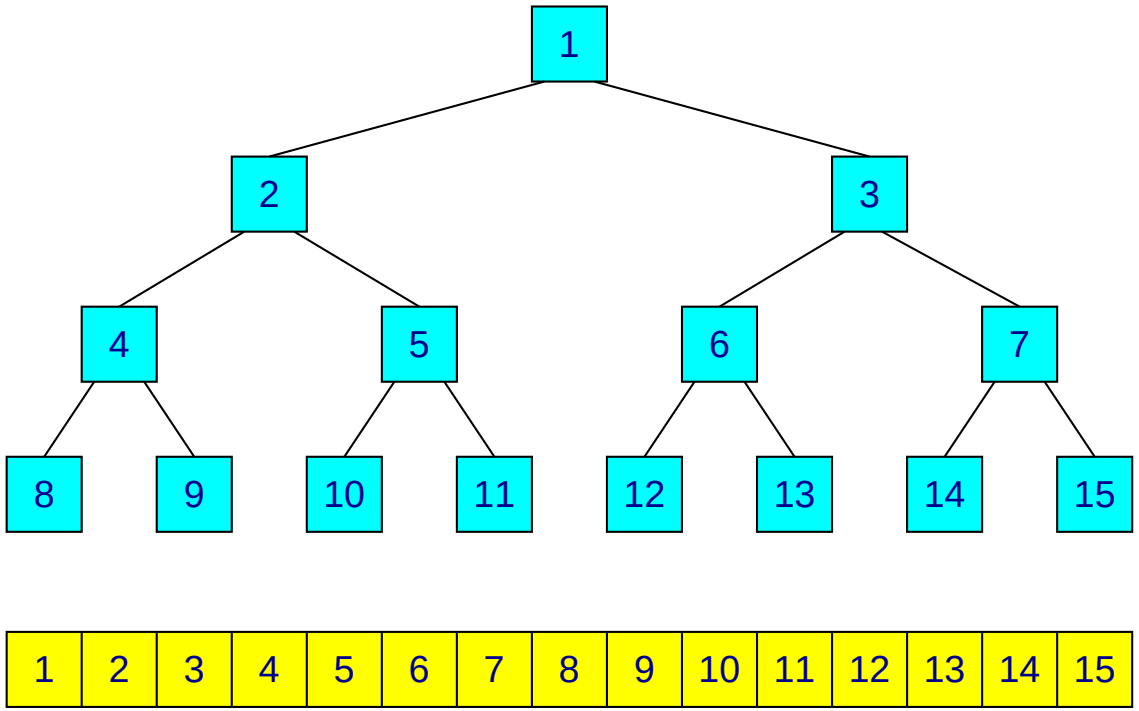
Složitost algoritmů

Efektivní uložení úplného binárního stromu v poli:



Složitost algoritmů

Efektivní uložení úplného binárního stromu v poli:



Potomci vrcholu s indexem i mají indexy $2i$ a $2i + 1$.
Rodič vrcholu s indexem i má index $\lfloor i/2 \rfloor$.

Složitost algoritmů

Halda (**heap**) — úplný binární strom uložený v poli A výše uvedeným způsobem, kde navíc pro každé $i = 1, 2, \ldots, n$ platí:

- pokud $2i \leq n$, pak $A[i] \leq A[2i]$
- pokud $2i + 1 \leq n$, pak $A[i] \leq A[2i + 1]$

Příklady využití haldy:

- třídící algoritmus **HeapSort**
- efektivní implementace **prioritní fronty** — umožňuje provádět většinu operací na této frontě s časovou složitostí v $\mathcal{O}(\log n)$, kde n je počet prvků ve frontě

Algoritmus: Vytvoření haldy z nesetříděného pole

```
CREATE-HEAP (A, n):
    i := ⌊n/2⌋
    while i ≥ 1 do
        j := i
        x := A[j]
        while 2 * j ≤ n do
            k := 2 * j
            if k + 1 ≤ n and A[k + 1] < A[k] then
                k := k + 1
            if x ≤ A[k] then break
            A[j] := A[k]
            j := k
        A[j] := x
        i := i − 1
```

Složitost algoritmů

Časová složitost algoritmu CREATE-HEAP:

- Rychlou a hrubou analýzou lehce zjistíme, že tato složitost je v $\mathcal{O}(n \log n)$ a v $\Omega(n)$:
  - Vnější cyklus se provede vždy $\lfloor n/2 \rfloor$ krát — počet průchodů je tedy v $\Theta(n)$.
  - Počet průchodů vnitřním cyklem v rámci jedné iterace vnějšího cyklu je očividně v $\mathcal{O}(\log n)$.
- Daleko méně zřejmé je, že celkový počet průchodů vnitřním cyklem (tj. dohromady přes všechny iterace vnějšího cyklu) je ve skutečnosti v $\mathcal{O}(n)$.

Celkově tedy dostáváme:

Časová složitost algoritmu CREATE-HEAP je v $\Theta(n)$.

Složitost algoritmů

Zdůvodnění toho, proč je počet průchodů vnitřním cyklem v $\mathcal{O}(n)$:

Předpokládejme pro jednoduchost, že všechny větve stromu jsou stejně dlouhé a mají délku h — platí tedy $n = 2^{h+1} - 1$.

Označme C_i, kde $0 \leq i < h$, celkový počet průchodů vnitřním cyklem, kdy je na začátku cyklu hodnota j v i-té vrstvě stromu (vrstvy jsou číslovány odshora $0, 1, 2, \ldots$).

Zjevně je celkový počet průchodů s dán vztahem

$$s = C_{h-1} + C_{h-2} + \cdots + C_0 = \sum_{i=0}^{h-1} C_i$$

Hodnotu C_i spočítáme jako celkový počet vrcholů ve vstvách $0, 1, \ldots, i$:

$$C_i = 2^0 + 2^1 + \cdots + 2^i = \sum_{k=0}^{i} 2^k = \frac{2^{i+1} - 1}{2 - 1} = 2^{i+1} - 1$$

Celkový součet pak spočítáme následovně:

$$s = \sum_{i=0}^{h-1} C_i = \sum_{i=0}^{h-1} (2^{i+1} - 1) = 2 \cdot \left(\sum_{i=0}^{h-1} 2^i \right) - \left(\sum_{i=0}^{h-1} 1 \right)$$

$$= 2 \cdot \frac{2^h - 1}{2 - 1} - h = 2^{h+1} - 2 - h = n - 1 - h = \mathcal{O}(n)$$

Analýza rekurzivních algoritmů

Rekurzivní algoritmy

Rekurzivní algoritmus je algoritmus, který převede řešení původního problému na řešení několika podobných problémů pro menší instance.

Obecné schéma rekurzivních algoritmů:

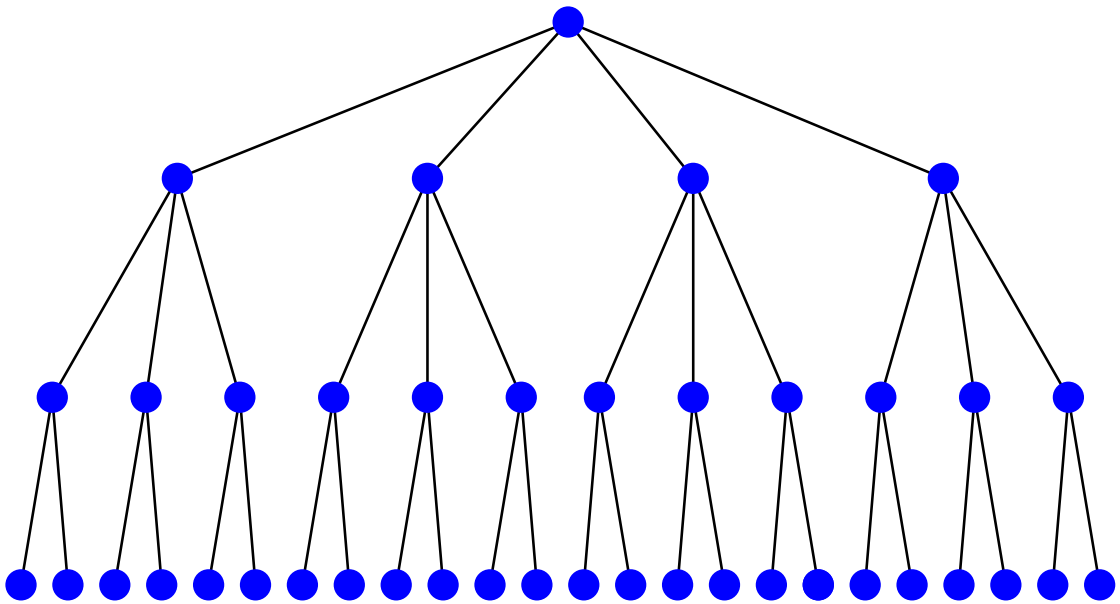
- Pokud se jedná o elementární případ, vyřeš ho přímo a vrať výsledek.
- V opačném případě vytvoř instance podproblémů.
- Zavolej sám sebe pro každou z těchto instancí.
- Z výsledků pro jednotlivé podproblémy slož řešení původního problému a vrať ho jako výsledek.

Poznámka: Instance podproblémů musí vždy být v nějakém smyslu menší než instance původního problému. Často (ne však vždy) se zmenšuje velikost instance.

Rekurzivní algoritmy

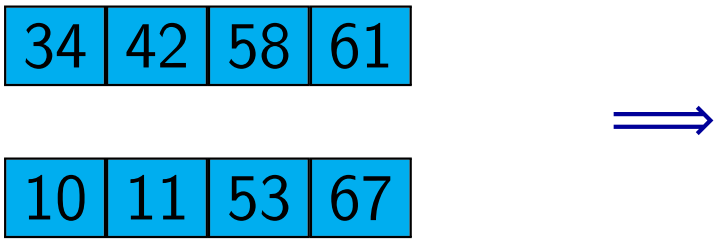
Výpočet rekurzivního algoritmu je možné znázornit jako strom:

- vrcholy stromu odpovídají jednotlivým podproblémům
- kořen je původní problém
- potomci vrcholu odpovídají podproblémům daného problému



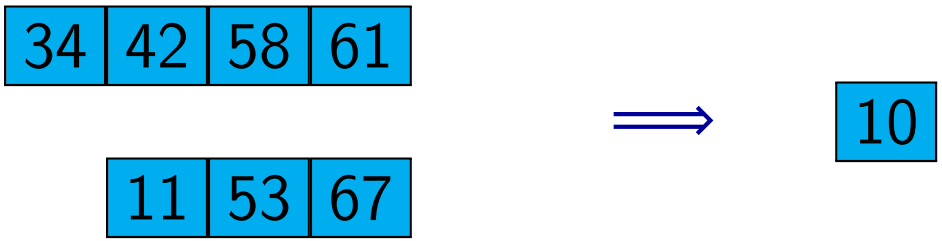
Příklad: Algoritmus MERGE-SORT.

Hlavní myšlenka algoritmu: Dvě setříděné posloupnosti snadno spojíme do jediné setříděné posloupnosti.
Pokud mají obě posloupnosti dohromady n prvků, vyžaduje tato operace n kroků.



Příklad: Algoritmus MERGE-SORT.

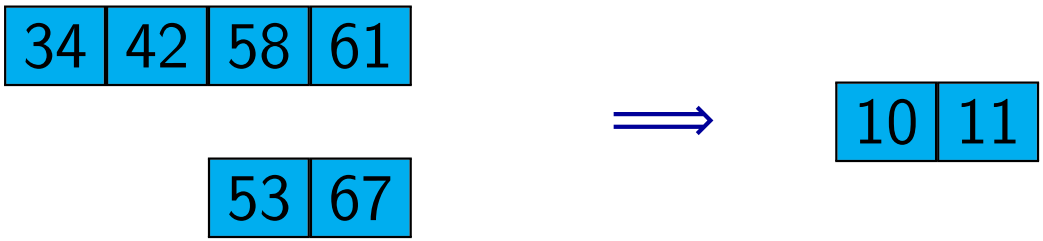
Hlavní myšlenka algoritmu: Dvě setříděné posloupnosti snadno spojíme do jediné setříděné posloupnosti.
Pokud mají obě posloupnosti dohromady n prvků, vyžaduje tato operace n kroků.



Rekurzivní algoritmy — Merge-Sort

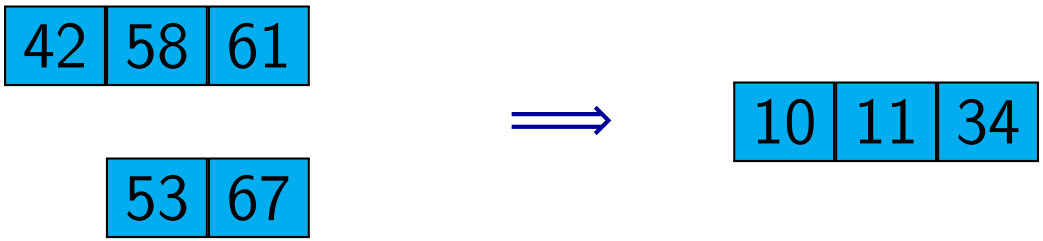
Příklad: Algoritmus MERGE-SORT.

Hlavní myšlenka algoritmu: Dvě setříděné posloupnosti snadno spojíme do jediné setříděné posloupnosti.
Pokud mají obě posloupnosti dohromady n prvků, vyžaduje tato operace n kroků.



Příklad: Algoritmus MERGE-SORT.

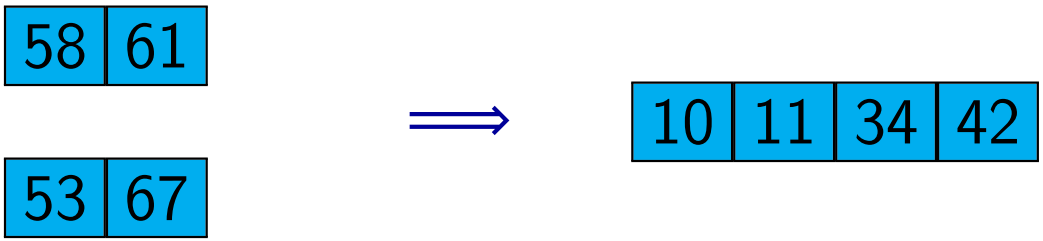
Hlavní myšlenka algoritmu: Dvě setříděné posloupnosti snadno spojíme do jediné setříděné posloupnosti.
Pokud mají obě posloupnosti dohromady n prvků, vyžaduje tato operace n kroků.



Rekurzivní algoritmy — Merge-Sort

Příklad: Algoritmus MERGE-SORT.

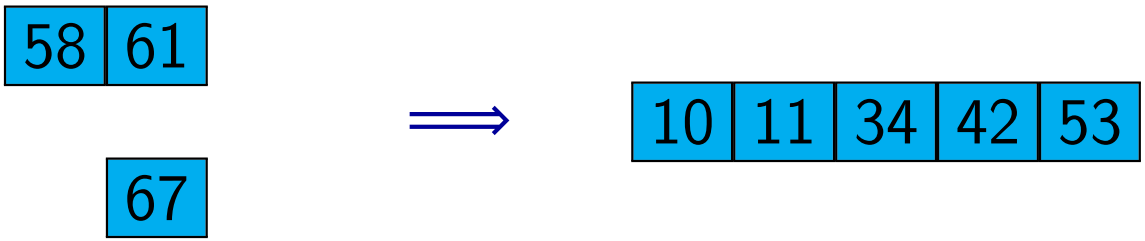
Hlavní myšlenka algoritmu: Dvě setříděné posloupnosti snadno spojíme do jediné setříděné posloupnosti.
Pokud mají obě posloupnosti dohromady n prvků, vyžaduje tato operace n kroků.



Rekurzivní algoritmy — Merge-Sort

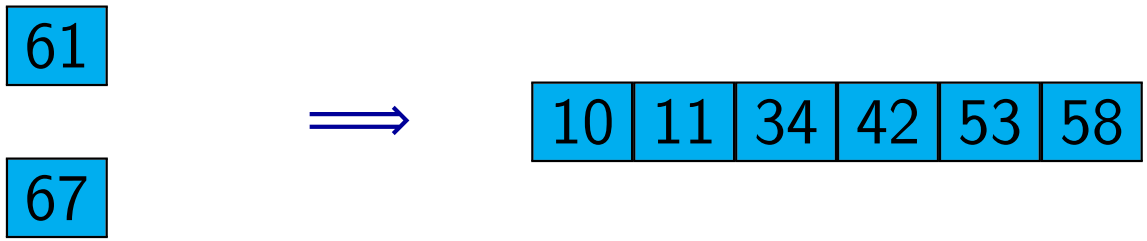
Příklad: Algoritmus MERGE-SORT.

Hlavní myšlenka algoritmu: Dvě setříděné posloupnosti snadno spojíme do jediné setříděné posloupnosti.
Pokud mají obě posloupnosti dohromady n prvků, vyžaduje tato operace n kroků.



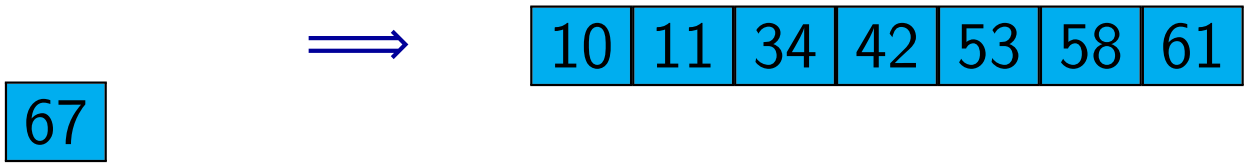
Příklad: Algoritmus MERGE-SORT.

Hlavní myšlenka algoritmu: Dvě setříděné posloupnosti snadno spojíme do jediné setříděné posloupnosti.
Pokud mají obě posloupnosti dohromady n prvků, vyžaduje tato operace n kroků.



Příklad: Algoritmus MERGE-SORT.

Hlavní myšlenka algoritmu: Dvě setříděné posloupnosti snadno spojíme do jediné setříděné posloupnosti.
Pokud mají obě posloupnosti dohromady n prvků, vyžaduje tato operace n kroků.



Rekurzivní algoritmy — Merge-Sort

Příklad: Algoritmus MERGE-SORT.

Hlavní myšlenka algoritmu: Dvě setříděné posloupnosti snadno spojíme do jediné setříděné posloupnosti.
Pokud mají obě posloupnosti dohromady n prvků, vyžaduje tato operace n kroků.

Rekurzivní algoritmy — Merge-Sort

Algoritmus: Merge sort

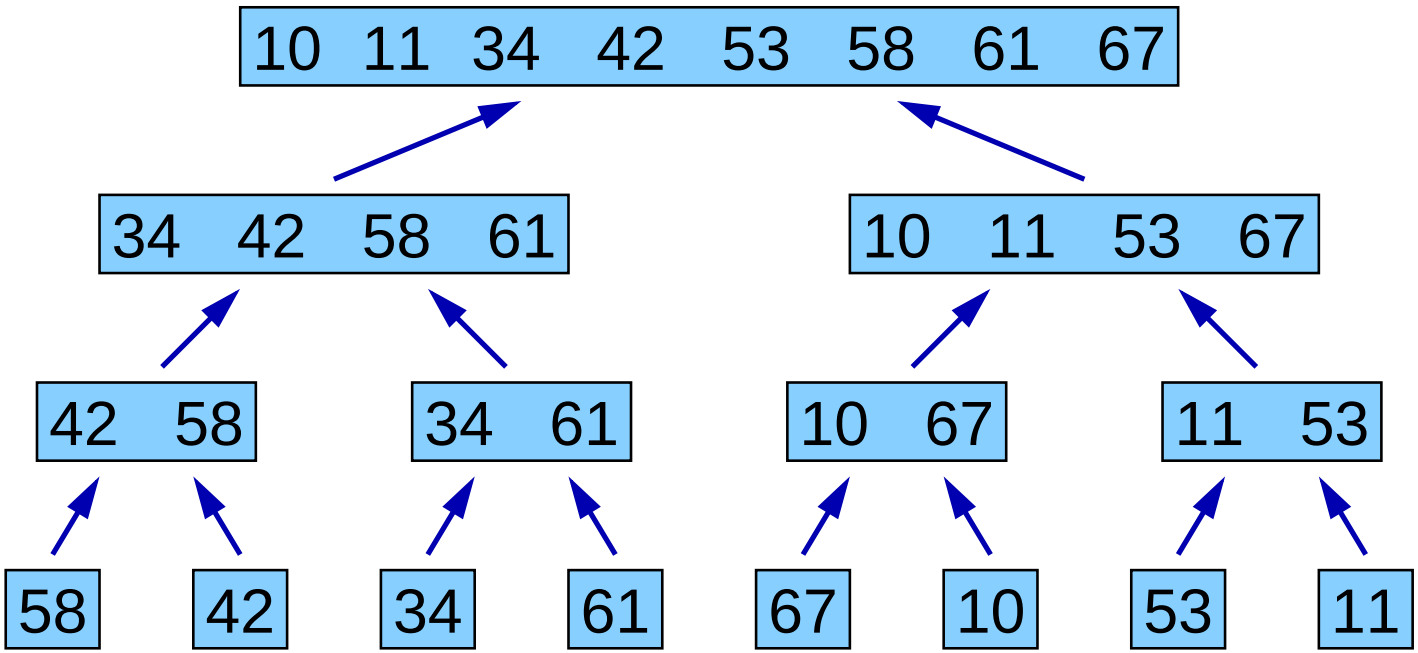
```
MERGE-SORT(A, p, r):
    if r − p > 1 then
        q := ⌊(p + r) / 2⌋
        MERGE-SORT(A, p, q)
        MERGE-SORT(A, q, r)
        MERGE(A, p, q, r)
```

Pro setřídění pole A, které obsahuje prvky $A[0], A[1], \cdots, A[n-1]$, zavoláme $\text{MERGE-SORT}(A, 0, n)$.

Poznámka: Procedura $\text{MERGE}(A, p, q, r)$ spojí setříděné posloupnosti uložené v $A[p \,..\, q-1]$ a $A[q \,..\, r-1]$ do jedné posloupnosti uložené v $A[p \,..\, r-1]$.

Rekurzivní algoritmy — Merge-Sort

Vstup: 58, 42, 34, 61, 67, 10, 53, 11



Strom rekurzivních volání má $\Theta(\log n)$ úrovní. Na každé úrovni se provede $\Theta(n)$ operací. Časová složitost algoritmu MERGE-SORT je $\Theta(n \log n)$.

The master theorem

Master theorem

Předpokládejme, že $a \geq 1$ a $b > 1$ jsou konstanty, že $f(n)$ je funkce a že funkce $T(n)$ je definována rekurentním předpisem

$$T(n) = a \cdot T(n/b) + f(n)$$

(kde n/b může být buď $\lfloor n/b \rfloor$ nebo $\lceil n/b \rceil$). Pak platí:

a. Pokud $f(n) \in \mathcal{O}(n^{\log_b a - \epsilon})$ pro nějakou konstantu $\epsilon > 0$, pak $T(n) = \Theta(n^{\log_b a})$.

b. Pokud $f(n) \in \Theta(n^{\log_b a})$, pak $T(n) = \Theta(n^{\log_b a} \log n)$.

c. Pokud $f(n) \in \Omega(n^{\log_b a + \epsilon})$ pro nějakou konstantu $\epsilon > 0$ a pokud $a \cdot f(n/b) \leq c \cdot f(n)$ pro nějakou konstantu $c < 1$ a všechna dostatečně velká n, pak $T(n) = \Theta(f(n))$.

The master theorem

Master theorem je možné použít pro analýzu složitosti těch rekurzivních algoritmů, kde:

- Řešení jednoho podproblému velikosti n, kde $n > 1$, se převede na řešení a podproblémů, z nichž každý má velikost n/b.
- Doba, která se stráví řešením jednoho podproblému velikosti n, nepočítaje v to dobu, která se stráví v rekurzivních voláních, je určená funkcí $f(n)$.

Příklad: Algoritmus MERGE-SORT: $a = 2$, $b = 2$, $f(n) \in \Theta(n)$

(v rámci jednoho volání — dva podproblémy, každý velikosti $n/2$, spojení dvou setříděných sekvencí v čase $\Theta(n)$)

Platí $f(n) \in \Theta(n^{\log_b a}) = \Theta(n)$, takže

$$T(n) \in \Theta(n^{\log_b a} \log n) = \Theta(n \log n).$$

The master theorem — násobení velkých čísel

Příklad: Řekněme, že chceme pracovat s přirozenými čísly, která jsou natolik velká, že se nevejdou do číselných datových typů, které máme k dispozici.

Například chceme pracovat s čísly, která mají 4096 bitů, ale operace s čísly, které máme k dispozici v daném programovacím jazyce, ve kterém pracujeme, umožňují přímo pracovat jen s čísly, která mají 32 nebo 64 bitů.

Přirozeným způsobem, jak to vyřešit, je uložit každé takové „velké“ číslo do pole příslušné velikosti, kde každý prvek tohoto pole bude „malé“ číslo velikosti, se kterou můžeme přímo pracovat.

The master theorem — násobení velkých čísel

„Velké“ číslo u tak můžeme uložit do pole U s prvky $U[0], \ldots, U[n-1]$, kde každý prvek bude představovat jedno „malé“ číslo v rozsahu $0, \ldots, q-1$, kde q je nějaký vhodně zvolený základ takový, že můžeme přímo pracovat s čísly v tomto rozsahu.

Bude platit

$$u = \sum_{i=0}^{n-1} U[i] \cdot q^i$$

Na takto uložené číslo se můžeme dívat tak, že jde o zápis čísla u v číselné soustavě o základu q, a prvky pole U představují jednotlivé „číslice“ tohoto zápisu.

Za velikost čísla u budeme považovat počet takovýchto „číslic“ nutných k jeho zápisu (tj. číslo n).

The master theorem — násobení velkých čísel

Dvě čísla velikosti n můžeme snadno sečíst v čase $\mathcal{O}(n)$ použitím standardního školního algoritmu pro sčítání.

Podobně můžeme začít uvažovat o problému **násobení** dvou přirozených čísel:

Násobení dvou přirozených čísel

Vstup: Číslo u uložené v poli U velikosti n
a číslo v uložené v poli V velikosti n.

Výstup: Číslo w uložené v poli W velikosti $2n$ takové, že $u \cdot v = w$.

Klasický školní algoritmus pro násobení bude mít časovou složitost $\Theta(n^2)$.

The master theorem — násobení velkých čísel

Místo tohoto klasického algoritmu můžeme uvažovat o **rekurzivním** algoritmu založeném na následující myšlence:

- Pokud jsou čísla velikosti 1 (tj. $n = 1$), tak je vynásobíme přímo.
- Pokud mají větší velikost (tj. $n > 1$), rozdělíme obě čísla na čísla zhruba poloviční velikosti, tj.

$$u = U_1 \cdot Q + U_0$$
$$v = V_1 \cdot Q + V_0$$

kde $Q = q^{\lceil n/2 \rceil}$.

Výsledek násobení $w = u \cdot v$ pak spočítáme jako

$$w = W_2 \cdot Q^2 + W_1 \cdot Q + W_0$$

kde

$$W_2 = U_1 \cdot V_1$$
$$W_1 = U_0 \cdot V_1 + U_1 \cdot V_0$$
$$W_0 = U_0 \cdot V_0$$

The master theorem — násobení velkých čísel

Problém násobení dvou čísel velikosti n tak převedeme na 4 problémy násobení čísel poloviční velikosti.

Pro vynásobení těchto menších čísel použijeme **rekurzi** — funkce zavolá sama sebe, přičemž jako argumenty jednotlivých volání budou předána tato menší čísla poloviční velikosti.

(Násobení mocninami Q je možno realizovat pomocí posunů o příslušný počet míst.)

The master theorem — násobení velkých čísel

Celkovou složitost algoritmu tak můžeme vyjádřit následujícím rekurzivním vztahem:

$$T(n) = 4 \cdot T(n/2) + \Theta(n)$$

Pro použití master theoremu máme $a = 4$, $b = 2$ a $f(n) \in \Theta(n)$:

- Platí $f(n) \in \mathcal{O}(n^{\log_b a - \epsilon})$, protože $n \in \mathcal{O}(n^{\log_2 4 - \epsilon}) = \mathcal{O}(n^{2-\epsilon})$ platí např. pro $\epsilon = 1$.
- Z master theoremu pak vyplývá, že
$$T(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 4}) = \Theta(n^2).$$

Celková časová složitost rekurzivního algoritmu pro násobení je tedy $\Theta(n^2)$, což je stejné jako v případě jednoduchého standardního algoritmu.

The master theorem — násobení velkých čísel

U rekurzivního algoritmu je však možné snížit počet násobení čísel poloviční velikosti ze 4 na 3:

- Nejprve spočítáme hodnoty W_2 a W_0 stejným způsobem jako předtím:

$$W_2 = U_1 \cdot V_1$$
$$W_0 = U_0 \cdot V_0$$

- Hodnotu W_1 pak spočítáme následovně:

$$W_1 = (U_0 + U_1) \cdot (V_0 + V_1) - W_0 - W_2$$

Ověření korektnosti:

$$\begin{aligned} W_1 &= (U_0 + U_1) \cdot (V_0 + V_1) - W_0 - W_2 \\ &= U_0 V_0 + U_0 V_1 + U_1 V_0 + U_1 V_1 - U_0 V_0 - U_1 V_1 \\ &= U_0 V_1 + U_1 V_0 \end{aligned}$$

The master theorem — násobení velkých čísel

Tento rekurzivní algoritmus pro násobení velkých čísel se označuje jako **Karacubovo násobení** (**Karatsuba multiplication**).

Podobně jako v předchozím případě můžeme vyjádřit složitost tohoto algoritmu rekurzivním vztahem

$$T(n) = 3 \cdot T(n/2) + \Theta(n)$$

A pomocí master theoremu (pro $a = 3$, $b = 2$ a $f(n) \in \Theta(n)$) odvodit

$$T(n) \in \Theta(n^{\log_2 3})$$

$\log_2 3$ je přibližně 1.5849625

Takže $T(n) \in \mathcal{O}(n^{1.59})$, což je lepší než $\Theta(n^2)$.

The master theorem — násobení velkých čísel

Poznámka:

- Existuje celá řada ještě efektivnějších algoritmů pro násobení velkých čísel, které jsou podobně jako Karacubovo násobení založeny na **rekurzivním přístupu**:
  - například různé varianty algoritmu Toom-Cook
- Nejefektivnější známé algoritmy pro násobení jsou založeny na **rychlé Fourierově transformaci** (**Fast Fourier transform**, **FFT**):
  - Schönhage–Strassen

  Algoritmus Schönhage-Strassen má složitost $\mathcal{O}(n \cdot \log n \cdot \log\log n)$.

  V roce 2019 byl objeven algoritmus (D. Harvey, J. van der Hoeven) se složitostí $\mathcal{O}(n \cdot \log n)$.

The master theorem — násobení matic

Příklad: Násobení čtvercových matic A a B velikosti $n \times n$ rekurzivním způsobem:

- Pro $n = 1$ se výsledek spočítá přímo.
- Pro $n > 1$ se každá z matic A a B rozloží na čtyři podmatice velikosti $(n/2) \times (n/2)$.
- Výsledek se poskládá pomocí sčítání a násobení těchto osmi menších matic. Pro násobení těchto menších matic se funkce zavolá rekurzivně.
- Přímočarý způsob vyžaduje 8 násobení matic velikosti $(n/2) \times (n/2)$.

  Máme tedy $a = 8$, $b = 2$, $f(n) \in \Theta(n^2)$.

  Platí $f(n) \in \mathcal{O}(n^{\log_b a - \epsilon})$, protože $n^2 \in \mathcal{O}(n^{\log_2 8 - \epsilon}) = \mathcal{O}(n^{3-\epsilon})$ platí např. pro $\epsilon = 1$.

  Takže $T(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 8}) = \Theta(n^3)$.

Tento postup tedy není lepší než standardní jednoduchý algoritmus pro násobení matic.

The master theorem — násobení matic

Existuje však chytrý způsob, jak výše uvedené provést komplikovanějším způsobem tak, že v rámci jednoho rekurzivního volání postačí rekurzivně volat funkci 7 krát
(za cenu většího počtu sčítání a odčítání).

Jedná se o tzv. **Strassenův algoritmus**.

Zde je $a = 7$, $b = 2$ a $f(n) \in \Theta(n^2)$.

Opět platí $f(n) \in \mathcal{O}(n^{\log_b a - \epsilon})$, protože $n^2 \in \mathcal{O}(n^{\log_2 7 - \epsilon})$ platí např. pro $\epsilon = 0.5$.

($\log_2 7$ je přibližně 2.80735)

Takže $T(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 7})$ a tedy $T(n) \in \mathcal{O}(n^{2.81})$.

The master theorem — důkaz

Důkaz master theoremu:

Pro jednoduchost se omezíme jen na případy, kdy $f(n) = n^c$ pro nějakou konstantu $c > 0$.

Rovněž pro jednoduchost předpokládejme, že n je mocninou čísla b, ať nemusíme řešit zaokrouhlování.

Představme si strom rekurzivních volání pro instanci velikosti n:

- Výška stromu je $\log_b n$.
- Počty vrcholů na jednotlivých úrovních jsou $a^0, a^1, \ldots, a^{\log_b n}$
- Čas, který se stráví v jednom vrcholu na úrovni i je

$$f\left(\frac{n}{b^i}\right) = \left(\frac{n}{b^i}\right)^c$$

The master theorem — důkaz

Platí tedy

$$T(n) = \sum_{i=0}^{\log_b n} a^i \cdot f\left(\frac{n}{b^i}\right) = \sum_{i=0}^{\log_b n} a^i \cdot \left(\frac{n}{b^i}\right)^c = n^c \cdot \sum_{i=0}^{\log_b n} \left(\frac{a}{b^c}\right)^i$$

Označme $q = a/b^c$. Je třeba rozlišit tři případy:

- $q > 1$ — tj. když platí $a > b^c$, neboli $c < \log_b a$
- $q = 1$ — tj. když platí $a = b^c$, neboli $c = \log_b a$
- $q < 1$ — tj. když platí $a < b^c$, neboli $c > \log_b a$

The master theorem — důkaz

Případ $q > 1$ — tj. když platí $a > b^c$, neboli $c < \log_b a$:

$$T(n) = n^c \cdot \sum_{i=0}^{\log_b n} \left(\frac{a}{b^c}\right)^i = n^c \cdot \frac{q^{\log_b n + 1} - 1}{q - 1} \in \Theta(n^c \cdot q^{\log_b n})$$

Platí

$$n^c \cdot q^{\log_b n} = n^c \cdot \left(\frac{a}{b^c}\right)^{\log_b n} = n^c \cdot n^{\log_b\left(\frac{a}{b^c}\right)}$$

$$= n^c \cdot n^{\log_b a - \log_b(b^c)} = n^{c + \log_b a - c} = n^{\log_b a}$$

Platí tedy $T(n) \in \Theta(n^{\log_b a})$.

Poznámka: Počet listů stromů (tj. podproblémů velikosti 1) je $a^{\log_b n} = n^{\log_b a}$.

Většina času se tedy tráví řešením těchto elementárních případů.

The master theorem — důkaz

Případ $q = 1$ — tj. když platí $a = b^c$, neboli $c = \log_b a$:

$$T(n) = n^c \cdot \sum_{i=0}^{\log_b n} \left(\frac{a}{b^c}\right)^i = n^c \cdot \sum_{i=0}^{\log_b n} 1 = n^c \cdot (\log_b n + 1) \in \Theta(n^{\log_b a} \log n)$$

Poznámka: V každé vrstvě stromu se stráví zhruba stejný čas $\Theta(n^{\log_b a})$.
Vrstev je celkem $\Theta(\log n)$.

The master theorem — důkaz

Případ $q < 1$ — tj. když platí $a < b^c$, neboli $c > \log_b a$:

$$T(n) = n^c \cdot \sum_{i=0}^{\log_b n} \left(\frac{a}{b^c}\right)^i < n^c \cdot \sum_{i=0}^{\infty} \left(\frac{a}{b^c}\right)^i = n^c \cdot \frac{1}{1-q} \in \mathcal{O}(n^c)$$

protože pro q, kde $0 < q < 1$, platí

$$\sum_{i=0}^{\infty} q^i = \lim_{z\to\infty} \sum_{i=0}^{z} q^i = \lim_{z\to\infty} \frac{q^{z+1}-1}{q-1} = \frac{1}{1-q}$$

Zjevně platí $T(n) \in \Omega(n^c)$ (protože už v samotném kořeni se stráví čas $\Theta(n^c)$), takže celkově platí $T(n) \in \Theta(n^c)$.

Poznámka: Většina času se v tomto případě stráví v kořeni stromu.