12 Délka výpočtu algoritmu

Mimo samotné správnosti výsledku vypočteného zapsaným algoritmem je ještě jedno neméně důležité hledisko k posouzení vhodnosti algoritmu k řešení zadané úlohy. Jedná se o čas, který algoritmus stráví výpočtem.

Asi netřeba argumentovat, že přehnaně dlouhá doba odezvy programu je každému uživateli nepříjemná. A co třeba v real-time systémech, kde si zdržení prostě nemůžeme dovolit!

Obligátní odpověď „kupme si rychlejší počítač“ bohužel není vždy řešením, jak při pokročilém studiu složitosti algoritmů sami poznáte. Mnohem větší potenciál zrychlení se skrývá v algoritmech samotných a jejich efektivním návrhu.

Stručný přehled lekce

* Délka výpočtu algoritmu, definice na našem deklarativním jazyce.
* Asymptotické určení délky výpočtu – asymptotické chování funkcí.
* Asymptotické odhady rekurentních vztahů.

12.1 O významu délky výpočtu algoritmu

Uvažme deklarativní jazyk Definice 9.1.

Definice: Délkou výpočtu výrazu F nad deklarací Δ rozumíme nejmenší přirozené k takové, že pro něj existuje $\mathbf{m} \in Num$ pro něž $F \mapsto^k \mathbf{m}$. □

(Když takové $\mathbf{m}$ neexistuje, klademe $k = \infty$.)

Jaká je délka výpočtu následujících výrazů?

* $\mathbf{3 + 4 - 5 * 6}$; □Tři kroky $\mathbf{3 + 4 - 5 * 6} \mapsto \mathbf{3 + 4 - 30} \mapsto \mathbf{3 + 0} \mapsto \mathbf{3}$.
* $\mathbf{3 + (5 - 4) * (6 \div 2)}$; □Tentokrát čtyři kroky
  $\mathbf{3 + (5 - 4) * (6 \div 2)} \mapsto \mathbf{3 + 1 * (6 \div 2)} \mapsto \mathbf{3 + 1 * 3} \mapsto \mathbf{3 + 3} \mapsto \mathbf{6}$.
* $\mathbf{2007}$; □Žádný krok, tj. $k = 0$.

Příklad 12.1. *Pro ukázku uvažme deklaraci Δ obsahující pouze rovnici*

$$f(x) = \text{if } x \text{ then } x * f(x - \mathbf{1}) \text{ else } \mathbf{1} \text{ fi}\,.$$

Věta. Pro každé $n \in \mathbb{N}$ je délka výpočtu výrazu $f(\mathbf{n})$ rovna $4n + 2$.□

Důkaz povedeme indukcí podle n:

- Báze $n = 0$. Platí $f(\mathbf{0}) \mapsto \text{if } \mathbf{0} \text{ then } \mathbf{0} * f(\mathbf{0} - \mathbf{1}) \text{ else } \mathbf{1} \text{ fi} \mapsto \mathbf{1}$, což jsou přesně 2 kroky, tj. $4 \cdot 0 + 2$.□
- Indukční krok. Nechť $n + 1 \equiv \mathbf{k}$. Pak

  $$f(\mathbf{k}) \mapsto \text{if } \mathbf{k} \text{ then } \mathbf{k} * f(\mathbf{k} - \mathbf{1}) \text{ else } \mathbf{1} \text{ fi} \mapsto \mathbf{k} * f(\mathbf{k} - \mathbf{1}) \mapsto \mathbf{k} * f(\mathbf{w})\,,$$

  kde $\mathbf{w} \equiv k - 1 = n$. □To jsou přesně 3 kroky. Podle I.P. je délka výpočtu výrazu $f(\mathbf{w})$ rovna $4n + 2$. Poté následuje ještě jeden poslední krok vynásobení $\mathbf{k}$. Celkem se provedlo $3 + 4n + 2 + 1 = 4(n + 1) + 2 = 4k + 2$ kroků. □

□

Počítat přesně nebo raději ne?

Jaký má smysl určení přesného počtu kroků algoritmu při dnešních CPU? Copak jsme dnes schopni jednoznačně říci, jak dlouho jedna instrukce CPU trvá?
Z druh strany, i když víme, že algoritmus A třeba potřebuje $2n$ kroků výpočtu a algoritmus B třeba potřebuje $3n$ kroků, je mezi nimi až takový rozdíl? Stačí, když B spustíme na dvakrát rychlejším počítači a poměr se hned obrátí. □

Obě tyto prakticky motivované úvahy nás povedou k poznání, že aditivní a multiplikativní faktory funkce počtu kroků algoritmu jsou vlastně zanedbatelné.

12.2 Asymptotické značení a odhady funkcí

Zajímá-li nás jen rychlost růstu funkce $f(n)$ v závislosti na n, zaměřujeme se především na tzv. *asymptotické chování* f při velkých hodnotách n.
V popisu f nás tedy nezajímají ani "drobné členy", ani konstanty, kterými je f násobena a které jen ovlivňují číselnou hodnotu $f(n)$, ale ne rychlost růstu.

Definice: Nechť $g : \mathbb{N} \to \mathbb{N}$ je daná funkce. Pro funkci $f : \mathbb{N} \to \mathbb{N}$ píšeme

$$f \in O(g)$$

pokud existují konstanty $A, B > 0$ takové, že

$$\forall n \in \mathbb{N} : f(n) \leq A \cdot g(n) + B\,.$$

V praxi se obvykle (i když matematicky méně přesně) píše místo $f \in O(g)$ výraz

$$f(n) = O(g(n))\,.$$

Znamená to, slovně řečeno, že funkce f *neroste rychleji* než funkce g. (I když pro malá n třeba může být $f(n)$ mnohem větší než $g(n)$.)

Definice: Píšeme $f \in \Omega(g)$, neboli

$$f(n) = \Omega(g(n))\,,$$

pokud $g \in O(f)$.
Dále píšeme $f \in \Theta(g)$, neboli

$$f(n) = \Theta(g(n))\,,$$

pokud $f \in O(g)$ a zároveň $f \in \Omega(g)$, neboli $g \in O(f)$.

Výraz $f(n) = \Theta(g(n))$ pak čteme jako "funkce f *roste stejně rychle* jako funkce g". □

Značení: O funkci $f(n)$ říkáme:

* $f(n) = \Theta(n)$ je *lineární* funkce,
* $f(n) = \Theta(n^2)$ je *kvadratická* funkce,
* $f(n) = \Theta(\log n)$ je *logaritmická* funkce,
* $f(n) = O(n^c)$ pro nějaké $c > 0$ je *polynomiální* funkce,
* $f(n) = \Theta(c^n)$ pro nějaké $c > 1$ je *exponenciální* funkce.

Příklad 12.2. *(opakovaný) Zjistěte, kolik znaků* `'x'` *v závislosti na celočíselné hodnotě* n *vstupního parametru* `n` *vypíše následující algoritmus.*

Algoritmus 12.3.

```
for  i ← 1,2,3,...,n-1,n  do
    for  j ← 1,2,3,...,i-1,i  do
        vytiskni  'x';
    done
done
```
□

Zatímco v Lekci 8 jsme trochu zdlouhavě indukcí dokazovali, že výsledkem je $\frac{1}{2}n(n+1)$ `'x'`, nyní si mnohem snadněji odvodíme, že počet `'x'` je $\Theta(n^2)$, což je dostačující asymptotická odpověď ve většině informatických aplikací.

Důkaz: Shora hned odhadneme, že každá z n iterací vnějšího cyklu vytiskne po $i \leq n$ znaků `'x'`, takže celkem je nejvýše n^2 `'x'`. Naopak zdola hned vidíme, že posledních $n/2$ iterací vnějšího cyklu vytiskne $i \geq n/2$ znaků `'x'`, takže celkem je alespoň $(n/2) \cdot (n/2) = n^2/4$ `'x'`. Z toho podle definice hned vyjde asymptotický odhad $\Theta(n^2)$. □

Příklad 12.4. *Příklady růstů různých funkcí.*

Funkce $f(n) = \Theta(n)$: pokud n vzroste na dvojnásobek, tak hodnota $f(n)$ taktéž vzroste (zhruba) na dvojnásobek. To platí jak pro funkci $f(n) = n$, tak i pro $1000000000n$ nebo $n + \sqrt{n}$, atd.

Funkce $f(n) = \Theta(n^2)$: pokud n vzroste na dvojnásobek, tak hodnota $f(n)$ vzroste (zhruba) na čtyřnásobek. To platí jak pro funkci $f(n) = n^2$, tak i pro $1000n^2 + 1000n$ nebo $n^2 - 99999999n - 99999999$, atd.

Naopak pro funkci $f(n) = \Theta(2^n)$: pokud n vzroste byť jen o 1, tak hodnota $f(n)$ už vzroste (zhruba) na dvojnásobek. To je obrovský rozdíl exponenciálních proti polynomiálním funkcím.

Pokud vám třeba funkce $999999n^2$ připadá velká, jak stojí ve srovnání s 2^n? Zvolme třeba $n = 1000$, kdy $999999n^2 = 999999000000$ je ještě rozumně zapsatelné číslo, ale $2^{1000} \simeq 10^{300}$ byste už na řádek nenapsali. Pro $n = 10000$ je rozdíl ještě mnohem výraznější! □

Rekurentní odhady

V tomto oddíle si uvedeme krátký přehled některých rekurentních vzorců, se kterými se můžete setkat při řešení časové složitosti (převážně rekurzivních) algoritmů.

Lema 12.5. *Nechť* $a_1, \ldots, a_k, c > 0$ *jsou kladné konstanty takové, že* $a_1 + \ldots + a_k < 1$, *a funkce* $T : \mathbb{N} \to \mathbb{N}$ *splňuje nerovnost*

$$T(n) \leq T(\lceil a_1 n \rceil) + T(\lceil a_2 n \rceil) + \ldots + T(\lceil a_k n \rceil) + cn\,.$$

Pak $T(n) = O(n)$.

Důkaz: Zvolme $\varepsilon > 0$ takové, že $a_1 + \ldots + a_k < 1 - 2\varepsilon$. Pak pro dostatečně velká n platí (i se zaokrouhlením nahoru) $\lceil a_1 n \rceil + \ldots + \lceil a_k n \rceil \leq (1 - \varepsilon)n$, řekněme pro všechna $n \geq n_0$. Dále zvolme dostatečně velké $d > 0$ tak, že $\varepsilon d > c$ a zároveň $d > \max\{\frac{1}{n}T(n) : n = 1, \ldots, n_0\}$.

Dále už snadno indukcí podle n dokážeme $T(n) \leq dn$ pro všechna $n \geq 1$:

- Pro $n \leq n_0$ je $T(n) \leq dn$ podle naší volby d.

- Předpokládejme, že $T(n) \leq dn$ platí pro všechna $n < n_1$, kde $n_1 > n_0$ je libovolné. Nyní dokážeme i pro n_1

$$T(n_1) \leq T(\lceil a_1 n_1 \rceil) + \ldots + T(\lceil a_k n_1 \rceil) + cn_1 \leq$$

$$\leq d \cdot \lceil a_1 n_1 \rceil + \ldots + d \cdot \lceil a_k n_1 \rceil + cn_1 \leq$$

$$\leq d \cdot (1 - \varepsilon) n_1 + cn_1 \leq dn_1 - (\varepsilon d - c) n_1 \leq dn_1 \,.$$

□

Lema 12.6. *Nechť $k \geq 2$ a $a_1, \ldots, a_k, c > 0$ jsou kladné konstanty takové, že $a_1 + \ldots + a_k = 1$, a funkce $T : \mathbb{N} \to \mathbb{N}$ splňuje nerovnost*

$$T(n) \leq T(\lceil a_1 n \rceil) + T(\lceil a_2 n \rceil) + \ldots + T(\lceil a_k n \rceil) + cn \,.$$

Pak $T(n) = O(n \cdot \log n)$.

V obecnosti je známo:

Lema 12.7. *Nechť $a \geq 1$, $b > 1$ jsou konstanty, $f : \mathbb{N} \to \mathbb{N}$ je funkce a pro funkci $T : \mathbb{N} \to \mathbb{N}$ platí rekurentní vztah*

$$T(n) \leq a \cdot T\left(\frac{n}{b}\right) + f(n)\,.$$

Pak platí:

* *Je-li $f(n) = O(n^c)$ a $c < \log_b a$, pak $T(n) = O(n^{\log_b a})$.*
* *Je-li $f(n) = \Theta(n^{\log_b a})$, pak $T(n) = O(n^{\log_b a} \cdot \log n)$.*
* *Je-li $f(n) = \Theta(n^c)$ a $c > \log_b a$, pak $T(n) = O(n^c)$.*

Důkaz tohoto obecného tvrzení přesahuje rozsah našeho předmětu. Všimněte si, že nikde ve výše uvedených řešeních nevystupují počáteční podmínky, tj. hodnoty $T(0), T(1), T(2), \ldots$ – ty jsou "skryté" v naší $O()$-notaci.

Dále v zápise pro zjednodušení zanedbáváme i necelé části argumentů, které mohou být zaokrouhlené.

Příklad 12.8. *Algoritmus merge-sort pro třídění čísel pracuje zhruba následovně:*

- *Danou posloupnost n čísel rozdělí na dvě (skoro) poloviny.*
- *Každou polovinu setřídí zvlášť za použití rekurentní aplikace merge-sort.*
- *Tyto dvě už setříděné poloviny "slije" (anglicky merge) do jedné setříděné výsledné posloupnosti.*

Jaký je celkový počet jeho kroků? □

Nechť na vstupu je n čísel. Při rozdělení na poloviny nám vzniknou podproblémy o velikostech $\lceil n/2 \rceil$ a $\lfloor n/2 \rfloor$ (pozor na necelé poloviny). Pokud počet kroků výpočtu označíme $T(n)$, pak rekurzivní volání trvají celkem

$$T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)\,.$$

Dále potřebujeme $c \cdot n$ kroků (kde c je vhodná konstanta) na slití obou částí do výsledného setříděného pole. Celkem tedy vyjde

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + cn \leq T(\lceil n/2 \rceil) + T(\lceil n/2 \rceil) + cn$$

a to už je tvar řešený v Lematu 12.6 pro $a_1 = a_2 = \frac{1}{2}$. Výsledek tedy je $T(n) = O(n \cdot \log n)$. □